Hibernate Session Commit Rollback Save Concepts

What is Hibernate Session?

A Session is used to get a **physical connection** with a database. The Session **object** is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object.

The session objects should not be kept **open** for a **long** time because they are not usually thread safe and they should be created and destroyed them as needed. The main **function** of the Session is to offer **create**, **read** and **delete** operations for instances of mapped entity classes. Instances may exist in one of the following three states at a given point in time:

transient: A new instance of a a persistent **class** which is not associated with a Session and has no representation in the database and no identifier value is considered **transient** by Hibernate.

persistent: You can make a **transient** instance persistent by associating it with a Session. A persistent instance has a representation in the database, an identifier value and is associated with a Session.

detached: Once we close the Hibernate Session, the persistent instance will become a detached instance.

A Session instance is serializable if its persistent classes are serializable. A typical transaction should use the following idiom:

```
Session session = factory.openSession();
Transaction tx = null;
try {
   tx = session.beginTransaction();
   // do some work
   ...
   tx.commit();
}
catch (Exception e) {
   if (tx!=null) tx.rollback();
   e.printStackTrace();
}finally {
   session.close();
}
```

If the Session **throws** an exception, the transaction must be rolled back and the session must be discarded.

QUESTION

I have been confused about `transaction.rollback` . Here is example pseudocode:

```
transaction = session.beginTransaction()
EntityA a = new EntityA();
session.save(a);
session.flush();
transaction.rollback();
```

What happens when this code works? Do I have the entity in the database or not?

ANSWER

When you call `session.save(a)` Hibernate basically remembers somewhere inside session that this **object** has to be saved. It can decide if he wants to issue `INSERT INTO...` immediately, some time later or on commit. This is a performance improvement, allowing Hibernate to **batch inserts** or avoid them if transaction is rolled back.

When you call `session.flush()` , Hibernate is forced to issue `INSERT INTO...` against the database. The entity is **stored** in the database, **but not yet committed**. Depending on transaction isolation level it won't be seen by other running transactions. But now the database *knows* about the record. What `session.flush()` does is to empty the internal SQL instructions cache, and execute it immediately to the database.

When you call `transaction.rollback()` , Hibernate rolls-back the database transaction. Database handles rollback, thus removing newly created object.

Now consider the scenario without `flush()` . First of all, you never touch the database so the performance is better and rollback is basically a no-op. On the other hand if transaction isolation level is `READ UNCOMMITTED` , other transactions can see inserted record even before commit/rollback. Without `flush()` this won't happen, unless Hibernate does not decide to `flush()` implicitly.

QUESTION

I googled a lot and read about `org.hibernate.Transaction.commit()` and `org.hibernate.Session.flush()` a lot, know purpose of each method, but still have a question.

Is it good practice to call `org.hibernate.Session.flush()` method by hand? As said in `org.hibernate.Session` docs,

```
Must be called at the end of a unit of work, before committing the transaction and closing
the session (depending on flush-mode, Transaction.commit() calls this method).
```

Could you explain me purpose of calling `org.hibernate.Session.flush()` by hand if `org.hibernate.Transaction.commit()` will call it automatically?

ANSWER

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**

QUESTION

When we are updating a record, we can use `session.flush()` with Hibernate. What's the need for **`flush() ?`**

ANSWER

Flushing the session forces Hibernate to synchronize the in-memory state of the `Session` with the database (i.e. to write changes to the database). By default, Hibernate will flush changes automatically for you:

- before some query executions
- when a transaction is committed

Allowing to explicitly flush the `Session` gives finer control that may be required in some circumstances (**to get an ID assigned, to control the size of the Session,...**).

QUESTION

If **FlushMode.AUTO** is set, will Hibernate flush my updated persistent **object** when I call `session.close()` ?

I know that `session.close()` does not normally flush the session but I'm not sure how **FlushMode.AUTO** affects this.

From the Docs:

`FlushMode.AUTO:`
`The Session is sometimes flushed before query execution in order to ensure that queries never return stale state. This is the default flush mode.`

Does this mean I can rely on Hibernate to verify my changes are flushed sometimes before my session is closed?

Small code example:

```
Session session = HibernateSessionFactory.getSession();
PersistedObject p = session.get(PersistedObject.class,id);
p.setSomeProperty(newValue);
session.close();
```

ANSWER

Will Hibernate flush my updated persistent **object** when calling session.close() (**using FlushMode.AUTO)?**

No it won't, and you should **use a transaction with well defined boundaries**. Quoting Non-transactional data access and the auto-commit mode (http://community.jboss.org/wiki/Non-transactionaldataaccessandtheauto-commitmode) : Working nontransactionally with Hibernate Look at the following code, which accesses the database without transaction boundaries:

```
Session session = sessionFactory.openSession();
session.get(Item.class, 123l);
session.close();
```

By default, in a Java SE environment with a JDBC configuration, this is what happens if you execute this snippet:

1. A new Session is opened. It doesn't obtain a database connection at this point.
2. The call to **get()** triggers an SQL **SELECT.** The Session now obtains a JDBC Connection from the connection pool. Hibernate, by default, immediately turns off the autocommit mode on this connection with setAutoCommit(**false).** This effectively starts a JDBC transaction!
3. The SELECT is executed inside this JDBC transaction. The Session is closed, and the connection is returned to the pool and released by Hibernate — Hibernate calls **close()** on the JDBC Connection.

What happens to the uncommitted transaction?

The answer to that question is, **"It depends!"** The JDBC specification doesn't say anything about pending transactions when **close()** is called on a connection. What happens depends on how the vendors implement the specification. With **Oracle** JDBC drivers, for example, the call to **close()** commits the transaction! Most other JDBC vendors take the sane route and roll back any pending transaction when the JDBC Connection **object** is closed and the resource is returned to the pool.

Obviously, this won't be a problem for the SELECT you've executed, but look at this variation:

```
Session session = getSessionFactory().openSession();
Long generatedId = session.save(item);
session.close();
```

This code results in an **INSERT** statement, executed inside a transaction that is **never** committed or **rolled** back. On Oracle, this piece of code inserts data permanently; in other databases, it may not. (This situation is slightly more complicated: The **INSERT** is executed only if the identifier generator requires it. For example, an identifier value can be obtained from a sequence without an **INSERT.** The persistent entity is then queued until flush-time insertion — which never happens in this code. An identity strategy requires an immediate **INSERT** for the value to be generated.)

Result: use **explicit** transaction demarcation.