**Q. What do you understand by automatic dirty checking in Hibernate?**

**A.** **Dirty checking**is a feature of hibernate that saves time and effort to update the database **when states of objects are modified inside a transaction**. All persistent objects are monitored by hibernate. It detects which objects have been modified and then calls update statements on all updated objects.  
  
Hibernate Session contains a ***PersistenceContext***object that maintains a cache of all the objects read from the database as a Map. So, when you modify an object within the same session, Hibernate compares the objects and triggers the updates when the session is flushed. The objects that are in the PersistenceContext are pesistent objects.  
  
**Q. How do you perform dirty checks for detached objects?**

**A.** When the session is closed, the ***PersistenceContext*** is lost and so is the cached copy, and the persistent object becomes a detached object. Detached objects can be passed all the way to the presentation layer.  When you reattach a detached object through *merge( ), update( ),* or *saveOrUpdate( )* methods, a new session is created with an empty PersistenceContext, hence there is nothing to compare against to perform the dirty check. Here is how you overcome this scenario with the help of the following annotation **selectBeforeUpdate = true**, by default it is set to false.  
  
  
To save the change to a detached object, you do something like

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | employee.setLastname("Smith");              // modifying a detached object  Session sess = sessionFactory.getSession(); // open a new session with an empty PersistenceContext  Transaction tx = sess.beginTransaction();   //begin a new transaction  sess.update(employee);//The detached object becomes a persistent object by reattaching  employee.setFirstName("John");              //modify the persistent object  tx.commit(); |

When the update() call is made, hibernate issues an SQL UPDATE statement. This happens irrespective of weather the object has changed after detaching or not. One way to avoid this UPDATE statement while reattaching is by setting the select-before-update= “true”. If this is set, hibernates tries to determine if the UPDATE needed by executing the SELECT statement. 

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Entity  @org.hibernate.annotations.Entity(selectBeforeUpdate = true)  @Table(name = "tbl\_employee")  public class Employee extends MyAppDomainObject implements Serializable {      .....  } |

In rare scenarios, where you are confident that a particular object can never be modified, hence you can tell hibernate that an UPDATE statement will never be needed by setting the following annotation 

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Entity  @org.hibernate.annotations.Entity(mutable = false)  @Table(name = "tbl\_employee")  public class Employee extends MyAppDomainObject implements Serializable {      .....  } |

Alternatively, if you want to decide if an object's state has changed or not and then decide if a redundant update call to be made or not, you can implement your own ***DirtyCheckInterceptor***by either implementing Hibenate's*Inteceptor*interface or extending the *EmptyInterceptor*class. The interceptor can be bootstrapped as shown below

|  |  |
| --- | --- |
|  | SessionFactory.openSession( new DirtyCheckInterceptor() ); |

and, Hibernate's ***FlushEntityEventListener's***onFlushEntity implementation calls the registered interceptor before making an update call.   
  
Another possible approach would be to clone the detached objects and store it somewhere in an HttpSession and then use that to populate the PersistenceContext when reattaching the object. For example, 

[?](http://java-success.blogspot.in/2013/01/hibernate-automatic-dirty-checking-of.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Session sess = sessionFactory.getSession();  PersistenceContext persistenceContext = session instanceof SessionImpl ? ((SessionImpl) session).getPersistenceContext(): null;    if (persistenceContext != null) {    addPreviouslyStoredEntitiesToPersistenceContext(persistenceContext, storedObjects);  } |

**Q. What do you understand by the terms optimistic locking versus pessimistic locking?**

**Optimistic locking** means a specific record in the database table is open for all users/sessions. Optimistic locking uses a strategy where you read a record, make a note of the version number and check that the version number hasn't changed before you write the record back. When you write the record back, you filter the update on the version to make sure that it hasn't been updated between when you check the version and write the record to the disk. If the record is dirty (i.e. different version to yours) you abort the transaction and the user can re-start it.  
  
You could also use other strategies like checking the timestamp or all the modified fields (this is useful for legacy tables that don't have version number or timestamp column). **Note**: The strategy to compare version numbers and timestamp will work well with detached hibernate objects as well. Hibernate will automatically manage the version numbers.   
  
In Hibernate, you can use either long number or Date for versioning 

|  |  |
| --- | --- |
|  | @Version  private long id; |

or 

|  |  |
| --- | --- |
|  | @Version  private Date version; |

and mark 

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Entity  @org.hibernate.annotations.Entity(selectBeforeUpdate = true, optimisticLock=OptimisticLockType.VERSION)  @Table(name = "tbl\_employee")  public class Employee extends MyAppDomainObject implements Serializable {      .....  } |

If you have a legacy table that does not have a version or timestamp column, then uses either 

[?](http://java-success.blogspot.in/2013/01/hibernate-automatic-dirty-checking-of.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Entity  @org.hibernate.annotations.Entity(selectBeforeUpdate = true, optimisticLock=OptimisticLockType.ALL)  @Table(name = "tbl\_employee")  public class Employee extends MyAppDomainObject implements Serializable {      .....  } |

for all fields and 

[?](http://java-success.blogspot.in/2013/01/hibernate-automatic-dirty-checking-of.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | @Entity  @org.hibernate.annotations.Entity(selectBeforeUpdate = true, optimisticLock=OptimisticLockType.DIRTY)  @Table(name = "tbl\_employee")  public class Employee extends MyAppDomainObject implements Serializable {      .....  } |

for dirty fileds only.   
  
**Pessimistic locking** means a specific record in the database table is open for read/write only for that current session. The other session users cannot edit the same because you lock the record for your exclusive use until you have finished with it. It has much better integrity than optimistic locking, but requires you to be careful with your application design to avoid deadlocks. In pessimistic locking, appropriate transaction isolation levels need to be set, so that the records can be locked at different levels. The general isolation levels are

* Read uncommitted isolation
* **Read committed isolation**
* **Repeatable read isolation**
* Serializable isolation

It can be dangerous to use "read uncommitted isolation" as it uses one transaction’s uncommitted changes in a different transaction. The "Serializable isolation” is used to protect phantom (**అవాస్తవిక విషయం**) reads, phantom reads are not usually problematic, and this isolation level tends to scale very poorly. So, if you are using pessimistic locking, then read committed and repeatable reads are the most common ones.