Transactions Lab - Solutions

Part 1 - JPA will not allow write operations in the database without transactions

In order to fix the exception, change the *@TransactionAttribute(NOT_SUPPORTED)* into *@TransactionAttribute(REQUIRED*) in the *TransactionEJB* class file. Clean and build and redeploy the project.

Part 2 - Treating the whole set of operations as a single Transaction

Open 2 browser tabs with the transaction application. Use one tab to initialise our 2 "bank accounts":

(1) *write(x,100);write(y,100);*

Now, let's imagine that transaction T is moving money from bank account x to bank account y, whereas transaction V wants to calculate the sum of all money in both accounts.

In one tab, let's run:

(2) *write(x,0);delay(**10**);write(y,200);*

Now switch to the other tab (you have **10** seconds to do that), and run the following:

(3) *read(x);read(y);*

You should see that x=0 and y=100, because the first transaction has not terminated yet, so it seems that £100 have disappeared. If you re-run (3) after 10s have passed from (2), you should see a different (consistent) output for the two bank accounts, that is x=0 and y=200.

Now go into the *ExecutorEJB* java file and change *@TransactionAttribute(NOT_SUPPORTED)* into *@TransactionAttribute(REQUIRED*). Clean and build the project. If you try to re-create the inconsistent retrieval problem now, you will notice that when executing transaction (3), the container will wait until (2) has ended.

Part 3 - Understanding the *em.flush* method

To better understand this section, we will follow the instructions provided in the Optional section.
After enabling fine grained logging, you will notice that when running the 2 transactions for the first time your Entity Manager will perform 2 SELECT and 2 UPDATE operations for T1 (one per value – x and y) and 2 SELECT operations for T2. T1 only requires finding entities in the DB once, they will already be loaded into the persistence context when they are read.
When you run the experiment for the second time, you will notice that no UPDATE operation is performed on the DB. This is because the value of the update is the same as the value stored in the database. In the first run, the EM blocked T2 since it was requesting a resource that had been updated by T1. In the second run, since x is not updated, the transaction is not blocked at all.

When trying to run the same experiment with a delay of 65s, the application throws an exception and the transaction is rolled back. As explained in the links provided, the reason for the exception is that T2 could not obtain a lock on resource x within the time requested, which by default is 60s.

In order to get a deadlock, let's open two tabs again and run the following transactions:

T1: w(x,10);d(10);w(y,10);
T2: w(y,20);d(10);w(x,20);

T1 will obtain a lock on x first, wait 10s and try to obtain a lock on y. T2, on the other hand, will obtain a lock on y, wait 10s and try to obtain a lock on x. Since both transactions will wait for a lock to be released until the other transaction has ended, they have entered into a deadlock state. As explained in one of the links provided, the container will wait for 20s after which Derby checks whether a transaction waiting to obtain a lock is involved in a deadlock. If a deadlock has occurred, and Derby chooses the transaction as a deadlock victim, Derby aborts the transaction.

After commenting the em.flush() call, the second transaction does not block because the UPDATE on the DB's resource is not performed until T1 has terminated.

Part 4 - Optimistic Locking
To get an Optimistic lock exception, try to run the following transactions into 2 different tabs:

T1: r(x);d(10);w(x,20);
T2: w(x,10);

T1 will not be able to terminate successfully because a *OptimisticLockException* will be thrown. In fact, resource x cannot be updated because it has changed or been deleted since it was last read.