LUXOFT TRAINING

JVA-000
Java Persistence with Hibernate

Module 5
Transaction Management

Objectives

- Understand the transaction term
- Learn how to control transactions in JPA
- Observer JPA facilities to handle concurrent reads/updates to entities

© Luxoft Training 2013

Transaction: What exactly the transaction is?

A transaction is a sequence of operations, performed as a single logical unit of work

The single logical unit of work must have the following properties in order to qualify it as a transaction:

- **Atomicity**

  A transaction must be an atomic unit of work, i.e., all of its data modifications are performed, or no modification is performed at all.

- **Consistency**

  After a transaction is completed, all data must be left in a consistent state. The written data must confirm to the defined rules such as constraints, triggers, cascades, etc.

© Luxoft Training 2013

Transaction: What exactly the transaction is?

- **Isolation**

  Modifications of a given transaction must be isolated from modifications made by other concurrent transactions. A transaction never recognizes data in an intermediate state, which was potentially caused by another concurrent transaction.

- **Durability**

  After a transaction has been completed, its effects are permanently stored in the system. Modifications persist even in the case of a system failure.

© Luxoft Training 2013

Controlling Transactions

Depending on the transactional type of the entity manager the transactions may be controlled either:

- Through JTA
- Through use of the resource-local `EntityTransaction` API

A **container-managed** entity manager **must be a JTA entity manager**. JTA entity managers are only specified for use in Java EE containers

An **application-managed** entity manager **may be** either a **JTA entity manager** or a **resource-local entity manager**.

© Luxoft Training 2013

Controlling Transactions

JTA entity manager participates in JTA transaction which is begun and committed externally

The `EntityTransaction` interface is used to control resource transactions on resource-local entity managers

Use `EntityManager.getTransaction()` to get n instance of `EntityTransaction`

© Luxoft Training 2013

Controlling Transactions

Methods of **`EntityTransaction`**:

- **`begin()`** - starts a resource transaction
- **`commit()`** - commits the current resource transaction, writing any unflushed changes to the database
- **`rollback()`** - roll back the current resource transaction
- **`setRollbackOnly()`** - marks the current transaction as read only *(commit cannot be applied)*
- **`getRollbackOnly()`** - determines whether the current transaction has been marked for rollback
- **`isActive()`** - indicates whether a resource transaction is in progress

© Luxoft Training 2013

Controlling Transactions

```
public class EmployeeManager {

    @PersistenceContext
    private EntityManager em;

    @Resource
    private UserTransaction tx;

    public void createEmployee(Employee employee) throws Exception {

        tx.begin();

        try {
            em.persist(employee);
            tx.commit();
        } catch (Exception error) {
            tx.rollback();
            throw error;
        }
    }

}
```

Reference to JPA EntityManager

Reference to JTA UserTransaction

Begin transaction

Modify data and commit the changes

... or rollback the chages in case of error

Controlling transaction using JTA interface

© Luxoft Training 2013

Controlling Transactions

```
public class EmployeeManager {

    @PersistenceContext
    private EntityManager em;

    public void createEmployee(Employee employee) throws Exception {

        EntityTransaction tx = em.getTransaction();

        tx.begin();

        try {
            em.persist(employee);
            tx.commit();
        } catch (Exception error) {
            tx.rollback();
            throw error;
        }
    }

}
```

Reference to JPA EntityManager

Use EntityTransaction to controll transaction

Begin transaction

Modify data and commit the changes

... or rollback the chages in case of error

Controlling transaction using JTA interface

© Luxoft Training 2013

Data Integrity and Consistency

Normal Expectations for Transactions:

- **Update Integrity**

  Changes cannot be successfully committed when the state read and modified is older than latest version in the database

- **Read Consistency**

  All records that the current transaction has read, but not changed, are still the latest versions in the database when the current transaction successfully commits

© Luxoft Training 2013

Concurrency: Locking

Update Integrity and Read Consistency are obtained by:

- Pessimistic locking
- Level of transaction isolation
- Optimistic checking on modified object (optimistic “lock”)

© Luxoft Training 2013

Concurrency: Optimistic Locking

Assumes that data won't be modified between read write operations

How it works:

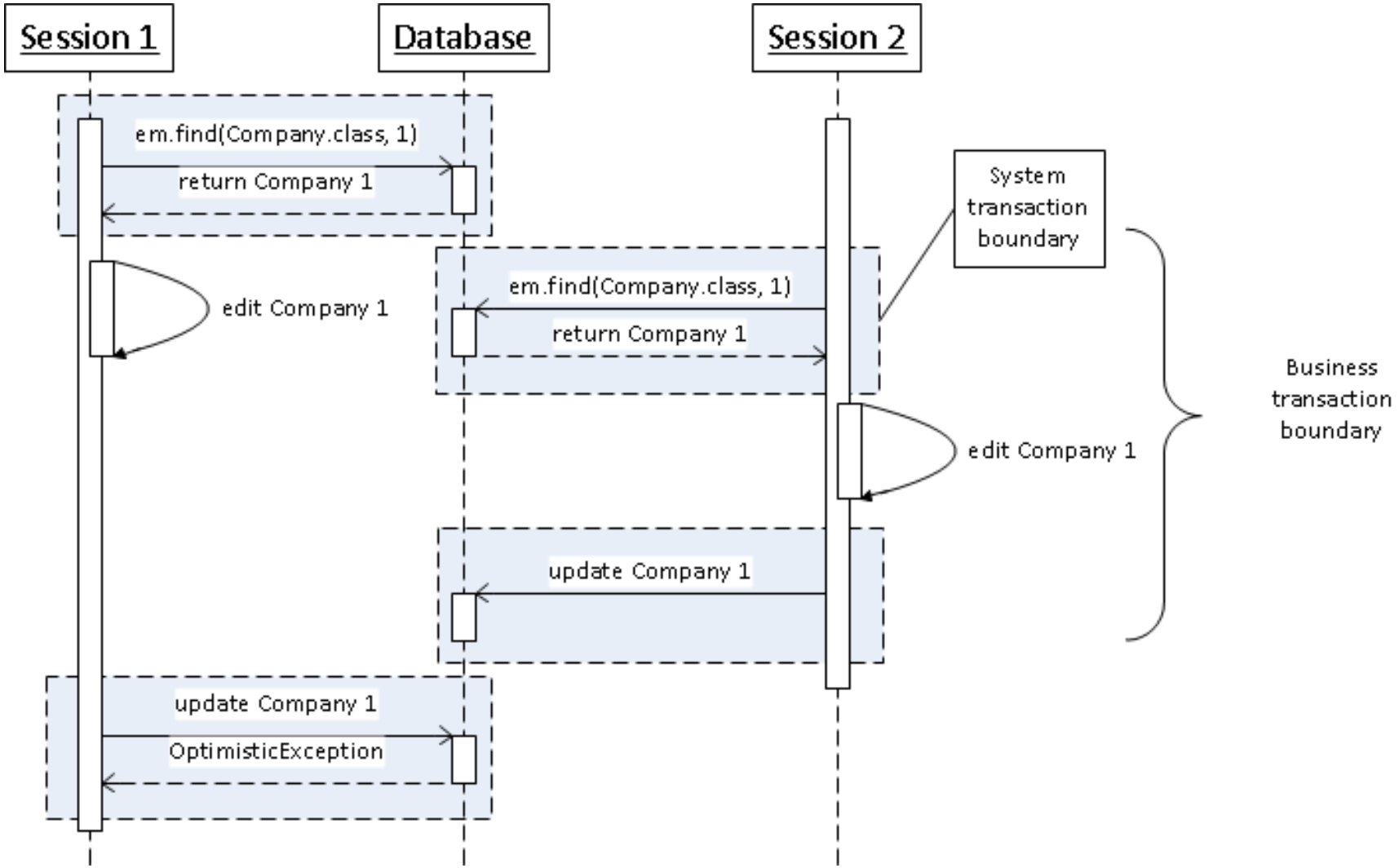
- **Reader** → reads version *(as part of entity state)*
- **Writer** → checks the version
- **Version mismatch** causes the **transaction to rollback**

Advantages:

- Maximum concurrency due to no database locks
- Less resources consumption on database

© Luxoft Training 2013

Concurrency: Optimistic Locking



How optimistic locking works

© Luxoft Training 2013

Concurrency: Optimistic Locking

Optimistic locking gets enabled if entity has version attribute that is:

- Annotated with **`@Version`**
- Has type either:
  - `int/long/etc`
  - `java.sql.Timestamp`
- Mapped into the primary table
- Should not be changed by user (in most of cases)

© Luxoft Training 2013

Concurrency: Optimistic Locking

Persistence provider:

- Updates version attribute automatically
- Examines version attribute if entity is modified
- Throws `OptimisticLockException` if object has stale version during:
  - Transaction commit: `Transaction.commit()`
  - Transaction flush: `Transaction().flush()`

© Luxoft Training 2013

Concurrency: Optimistic Locking

```java
@Entity
public class Company {
    @Id @GeneratedValue
    private long id;

    private String name;

    @Version
    private int version;

    public long getId() { return id; }
    public void setId(long id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

Version attribute used to insure data integrity for merge operations and to handle optiomistic locks

Entity should have `@Version` attribute to enable optimistic locking

© Luxoft Training 2013

Concurrency: Optimistic Locking

```
EntityManager em1 = emf.createEntityManager();
EntityManager em2 = emf.createEntityManager();
```

```
em1.getTransaction().begin();
Company company1 = em1.find(Company.class, 1);
```

Session 1: User gets the entity.

```
em2.getTransaction().begin();
Company company2 = em2.find(Company.class, 1);
company2.setName("Microsoft");
em2.getTransaction().commit();
```

Session 2: Another user gets the same entity and modifies it.

```
company1.setName("IMB");
em1.getTransaction().commit();
```

Session 1: The first user tries to modify entity that has already expired state. The RollbackException is thrown: cause is OptimisticException during entity modification.

If you attempt to modify entity with expired version the `OptimisticException` will be thrown which leads to transaction rollbacks

© Luxoft Training 2013

Concurrency: Pessimistic Locking

Readers obtain exclusive write locks on records read

- By using select-for-update
- Prevents concurrent reads
- Lock is released when transaction commits
- Reduces maximum achievable concurrency

Deadlocks are possible *(application is responsible to avoid them)*

Pessimistic transactions tend to fail slowly during reads *(due to slow deadlock detection)*

© Luxoft Training 2013

Concurrency: Pessimistic Locking

When entity instance is locked using pessimistic locking, the persistence provider locks:

- row that correspond to the persistent state of that instance
- rows in additional tables in case of joined inheritance or secondary table usage

© Luxoft Training 2013

Concurrency: LockModeType

Lock modes can be specified by means of passing a LockModeType argument to one of the **EntityManager** methods:

- **LockModeType.READ**
- **LockModeType.WRITE**
- **LockModeType.OPTIMISTIC**
- **LockModeType.OPTIMISTIC_FORCE_INCREMENT**
- **LockModeType.PESSIMISTIC_READ**
- **LockModeType.PESSIMISTIC_WRITE**
- **LockModeType.PESSIMISTIC_FORCE_INCREMENT**
- **LockModeType.NONE**

© Luxoft Training 2013

Concurrency: `LockModeType`

`NONE` – no locks applied

`OPTIMISTIC`:

- Used for **optimistic locking**
- **Prevents** having **Dirty Reads** and **Non-repeatable Reads**

`OPTIMISTIC_FORCE_INCREMENT`:

- Forces **version increment** even there are no updates to entity (additionally to **`OPTIMISTIC`**)

`READ` – synonym of **`OPTIMISTIC`**

`WRITE` – synonym of **`OPTIMISTIC_FORCE_INCREMENT`**

© Luxoft Training 2013

Concurrency: `LockModeType`

`PESSIMISTIC_READ`:

- Other transactions **can concurrently read** the entity, **but cannot concurrently update** it

`PESSIMISTIC_WRITE`:

- Other transactions **cannot concurrently read or write** the entity

`PESSIMISTIC_FORCE_INCREMENT`:

- Forces **version increment** even there are no updates to entity (additionally to **`PESSIMISTIC_WRITE`**)

© Luxoft Training 2013

Concurrency: LockModeType

	PESSIMISTIC_READ	PESSIMISTIC_WRITE
Lock type	Shared/Read Lock	Exclusive Lock
Is ReadOnly without additional locks	Yes	No
Allows dirty reads	No	No
Allows non-repeatable reads	No	No
How to update the entity state	Obtain PESSIMISTIC_WRITE	Allowed
When to use	You want to ensure no dirty reads or non-repeatable reads are possible	When there is a high likelihood of deadlock or update failure among concurrent updating transactions

© Luxoft Training 2013

Concurrency: Locking

How to lock the entity:

Call `EntityManager.lock()` with lock type as parameter

```
EntityManager em = createEntityManager();
em.find(Company.class, 1L, LockModeType.PESSIMISTIC_WRITE);
```

By call `EntityManager.lock()` method:

```
EntityManager em = createEntityManager();
Company company = em.find(Company.class, 1L);
em.lock(company, LockModeType.PESSIMISTIC_WRITE);
```

© Luxoft Training 2013

LUXOFT TRAINING

Thank you for your attention!

Questions?