EJB - TRANSACTIONS

A transaction is a single unit of work items which follows the ACID properties. ACID stands for Atomic, Consistent,Isolated and Durable.

- **Atomic** - If any of work item fails, the complete unit is considered failed. Success meant all items executes successfully.
- **Consistent** - A transaction must keep the system in consistent state.
- **Isolated** - Each transaction executes independent of any other transaction.
- **Durable** - Transaction should survive system failure if it has been executed or committed.

EJB Container/Servers are transaction servers and handles transactions context propagation and distributed transactions. Transactions can be managed by the container or by custom code handling in bean's code.

- **Container Managed Transactions** - In this type, container manages the transaction states.
- **Bean Managed Transactions** - In this type, developer manages the life cycle of transaction states.

Container Managed Transactions

EJB 3.0 has specified following attributes of transactions which EJB containers implement.

- **REQUIRED** - Indicates that business method has to be executed within transaction otherwise a new transaction will be started for that method.
- **REQUIRES_NEW** - Indicates that a new transaction is to be started for the business method.
- **SUPPORTS** - Indicates that business method will execute as part of transaction.
- **NOT_SUPPORTED** - Indicates that business method should not be executed as part of transaction.
- **MANDATORY** - Indicates that business method will execute as part of transaction otherwise exception will be thrown.
- **NEVER** - Indicates if business method executes as part of transaction then an exception will be thrown.

Example

```
package com.tutorialspoint.txn.required;

import javax.ejb.*

@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class UserDetailBean implements UserDetailRemote {

   private UserDetail;

   @TransactionAttribute(TransactionAttributeType.REQUIRED)
   public void createUserDetail() {
      //create user details object
   }
}
```

createUserDetail business method is made Required using Required annotation.

```
package com.tutorialspoint.txn.required;

import javax.ejb.*

@Stateless
public class UserSessionBean implements UserRemote {

   private User;

   @EJB
   private UserDetailRemote userDetail;

   public void createUser() {
      //create user
      //...
      //create user details
      userDetail.createUserDetail();
   }
}
```

createUser business method is using createUserDetail. If exception occured during createUser call and User object is not created then UserDetail object will also not be created.

Bean Managed Transactions

In Bean Managed Transactions, Transactions can be managed by handling exceptions at application level. Following are the key points to be considered

- **Start** - When to start a transaction in a business method.
- **Sucess** - Identify success scenario when a transaction is to be committed.
- **Failed** - Identify failure scenario when a transaction is to be rollback.

Example

```
package com.tutorialspoint.txn.bmt;

import javax.annotation.Resource;
import javax.ejb.Stateless;
import javax.ejb.TransactionManagement;
import javax.ejb.TransactionManagementType;
import javax.transaction.UserTransaction;

@Stateless
@TransactionManagement(value=TransactionManagementType.BEAN)
public class AccountBean implements AccountBeanLocal {

   @Resource
   private UserTransaction userTransaction;

   public void transferFund(Account fromAccount, double fund ,
      Account toAccount) throws Exception{

      try{
         userTransaction.begin();

         confirmAccountDetail(fromAccount);
         withdrawAmount(fromAccount,fund);

         confirmAccountDetail(toAccount);
         depositAmount(toAccount,fund);

         userTransaction.commit();
      }catch (InvalidAccountException exception){
         userTransaction.rollback();
      }catch (InsufficientFundException exception){
         userTransaction.rollback();
```

```
      }catch (PaymentException exception){
         userTransaction.rollback();
      }
   }

   private void confirmAccountDetail(Account account)
      throws InvalidAccountException {
   }

   private void withdrawAmount() throws InsufficientFundException {
   }

   private void depositAmount() throws PaymentException{
   }
}
```

In this example, we made use of **UserTransaction** interface to mark beginning of transaction using **userTransaction.begin** method call. We mark completion of transaction by using **userTransaction.commit** method and if any exception occured during transaction then we rollback the complete transaction using **userTransaction.rollback** method call.