3. Java Persistence API

5. Transaction Management

Database Transaction

- A database transaction is a sequence of actions that are treated as a single unit of work
- These actions should either complete entirely or take no effect at all
- Transaction management is an important part of RDBMS oriented enterprise applications to ensure data integrity and consistency.

ACID (1 of 2)

- Atomicity. A transaction should be treated as a single unit of operation which means either the entire sequence of operations is successful or unsuccessful
- Consistency. This represents the consistency of the referential integrity of the database, unique primary keys in tables etc

ACID (2 of 2)

- Isolation. There may be many transactions processing with the same data set at the same time, each transaction should be isolated from others to prevent data corruption
- Durability. Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure.

Spring Transaction Management

- Spring framework provides an abstract layer on top of different underlying transaction management APIs
- Local transactions are specific to a single transactional resource like a JDBC connection
- Global transactions can span multiple transactional resources like transaction in a distributed system

Local Transactions

- Local transaction management can be useful in a centralized computing environment where application components and resources are located at a single site, and transaction management only involves a local data manager running on a single machine
- Local transactions are easier to be implemented

Global Transactions

- Global transaction management is required in a distributed computing environment where all the resources are distributed across multiple systems
- A global transaction is executed across multiple systems, and its execution requires coordination between the global transaction management system and all the local data managers of all the involved systems

Programmatic vs. Declarative

- Spring supports two types of transaction management:
- Programmatic transaction management: you have manage the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain
- Declarative transaction management: you separate transaction management from the business code. You only use annotations or XML based configuration to manage the transactions

Programmatic vs. Declarative

- Declarative transaction management is preferable over programmatic transaction management

Spring Transaction Abstractions

- The key to the Spring transaction abstraction is defined by PlatformTransactionManager interface in the org.springframework.transaction package:

```
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(TransactionDefinition
    definition) throws TransactionException;
    void commit(TransactionStatus status) throws
   TransactionException;
    void rollback(TransactionStatus status) throws
   TransactionException;
}
```

PlatformTransactionManager

- getTransaction - returns a currently active transaction or create a new one, according to the specified propagation behavior
- commit - commits the given transaction, with regard to its status
- rollback - performs a rollback of the given transaction

TransactionDefinition

- Is the core interface of the transaction support in Spring and it is defined as below:

```
public interface TransactionDefinition {
    int getPropagationBehavior();
    int getIsolationLevel();
    String getName();
    int getTimeout();
    boolean isReadOnly();
}
```

TransactionDefinition Methods

- getPropagationBehavior - returns the propagation behavior
- getIsolationLevel - returns the degree to which this transaction is isolated from the work of other transactions
- getName - returns the name of the transaction
- getTimeout - returns the time in seconds in which the transaction must complete
- isReadOnly - returns whether the transaction is read-only.

Isolation Level (1 of 2)

- TransactionDefinition.ISOLATION_DEFAULT - the default isolation level
- TransactionDefinition.ISOLATION_READ_COMMITTED - indicates that dirty reads are prevented; non-repeatable reads and phantom reads can occur
- TransactionDefinition.ISOLATION_READ_UNCOMMITTED -dirty reads, non-repeatable reads and phantom reads can occur

Isolation Level (2 of 2)

- TransactionDefinition.ISOLATION_REPEATABLE_READ - dirty reads and non-repeatable reads are prevented; phantom reads can occur
- TransactionDefinition.ISOLATION_SERIALIZABLE - dirty reads, non-repeatable reads and phantom reads are prevented

Propagation Types (1 of 2)

- TransactionDefinition.PROPAGATION_MANDATORY - support a current transaction; throw an exception if no current transaction exists
- TransactionDefinition.PROPAGATION_NESTED - execute within a nested transaction if a current transaction exists
- TransactionDefinition.PROPAGATION_NEVER - do not support a current transaction; throw an exception if a current transaction exists
- TransactionDefinition.PROPAGATION_NOT_SUPPORTED - do not support a current transaction; rather always execute non-transactionally

Propagation Types (2 of 2)

- TransactionDefinition.PROPAGATION_REQUIRED - Support a current transaction; create a new one if none exists
- TransactionDefinition.PROPAGATION_REQUIRES_NEW - create a new transaction, suspending the current transaction if one exists
- TransactionDefinition.PROPAGATION_SUPPORTS - support a current transaction; execute non-transactionally if none exists
- TransactionDefinition.TIMEOUT_DEFAULT - use the default timeout of the underlying transaction system, or none if timeouts are not supported

TransactionStatus interface

- Provides a simple way for transactional code to control transaction execution and query transaction status

```
public interface TransactionStatus extends SavepointManager {
    boolean isNewTransaction();
    boolean hasSavepoint();
    void setRollbackOnly();
    boolean isRollbackOnly();
    boolean isCompleted();
}
```

TransactionStatus Methods

- hasSavepoint - returns whether this transaction internally carries a savepoint, that is, has been created as nested transaction based on a savepoint
- isCompleted - returns whether this transaction has already been committed or rolled back
- isNewTransaction - returns true in case the present transaction is new
- isRollbackOnly - returns whether the transaction has been marked as rollback-only
- setRollbackOnly - sets the transaction rollback-only

Declarative Transaction Management

- This approach allows you to manage the transaction with the help of configuration instead of hard coding in your source code

1. So you can separate transaction management from the business code by using annotations or XML based configuration to manage the transactions
2. The bean configuration will specify the methods to be transactional

Configuring Transaction Management

```
<bean id="transactionManager"
    class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory"
        ref="entityManagerFactory"/>
</bean>

<tx:annotation-driven transaction-manager="transactionManager"/>
```

Using @Transactional

- You can place the @Transactional annotation before a class definition, or a *public* method on a class
- A transaction begins before method annotated with @Transactional. It commits after method ends normally, and rollbacks if RuntimeException occurs.
- All methods for class annotated with @Transactional are transactional.

@Transactional Attributes

- propagation (Propagation.REQUIRED by default)
- Isolation (Isolation.DEFAULT by default)
- timeout (TransactionDefinition.TIMEOUT_DEFAULT)
- readonly
- rollbackFor
- rollbackForClassName
- noRollbackFor
- noRollbackForClassName

Exercise: Insert New Customer

- Insert new record to the CUSTOMER DB table – this problem was solved in P322AddCustomer project

New Save Interface Method

```
package com.bionic.edu;
public interface CustomerDao {
    public Customer findById(int id);
    public void save(Customer customer);
}
public interface CustomerService {
    public Customer findById(int id);
    public void save(Customer customer);
}
```

Save DAO Implementation

```
@Repository
public class CustomerDaoImpl implements CustomerDao{
    @PersistenceContext
    private EntityManager em;
    public Customer findById(int id){
        . . . . . . . . .
    }
    public void save(Customer customer){
        em.persist(customer);
    }
}
```

Save Service Implementation

```
@Named
public class CustomerServiceImpl implements CustomerService{
    @Inject
    private CustomerDao customerDao;
    public Customer findById(int id) {
        return customerDao.findById(id);
    }
    @Transactional
    public void save(Customer customer){
    customerDao.save(customer);
    }
}
```

Example: Payment of a New Customer

- The task is to add a payment of a new customer.

- The problem is that you need to save new customer's id in a Payment entity before the latter is saved.

PaymentDaoImpl Class

```
package com.bionic.edu;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import org.springframework.stereotype.Repository;
@Repository
public class PaymentDaoImpl implements PaymentDao{
    @PersistenceContext
    private EntityManager em;
    public void save(Payment p){
        em.persist(p);
    }
}
```

CustomerServiceImpl Class

```
    @Transactional
    public void add(Customer c, Payment p){
     save(c);
     p.setCustomerId(c.getId());
     paymentDao.save(p);
    }
}
```

Output

- INSERT on table 'PAYMENT' caused a violation of foreign key constraint 'CUSTOMER_FK' for key (0).

- Only external method calls can start transaction –**any self-invocation calls will not start any transaction**

- See 542AddCustPayment project for the full text

CustomerServiceImpl Class

```
package com.bionic.edu;
import javax.inject.Inject;
import javax.inject.Named;
import org.springframework.transaction.annotation.Transactional;
@Named
public class CustomerServiceImpl implements CustomerService{
    @Inject
    private CustomerDao customerDao;
    @Transactional
    public void save(Customer c){ customerDao.save(c); } }
```

PaymentServiceImpl Class

```
@Named
public class PaymentServiceImpl  implements PaymentService{
    @Inject
    private PaymentDao paymentDao;
    @Inject
    private CustomerService customerService;
    @Transactional
    public void add(Customer c, Payment p){
     customerService.save(c);
     p.setCustomerId(c.getId());
     paymentDao.save(p);
    }}
```

Output

- INSERT on table 'PAYMENT' caused a violation of foreign key constraint 'CUSTOMER_FK' for key (0).

- The reason is that propagation value of @Transactional annotation is REQUERED by default – so transaction for customerService.save(c) method will be commited along with paymentService.add(Customer c, Payment p) method

- See P362AddCustPayment project for the full text

CustomerServiceImpl Class

```
package com.bionic.edu;
import javax.inject.Inject;
import javax.inject.Named;
import org.springframework.transaction.annotation.Transactional;
@Named
public class CustomerServiceImpl implements CustomerService{
    @Inject
    private CustomerDao customerDao;
    @Transactional(propagation=Propagation.NESTED)
    public void save(Customer c){ customerDao.save(c); } }
```

Output

- JpaDialect does not support savepoints - check your JPA provider's capabilities

- The reason is that JPA doesn't support nested transactions
- Nested transactions are only supported on JDBC level directly

- See P363AddCustPayment project for the full text

CustomerServiceImpl Class

```
package com.bionic.edu;
import javax.inject.Inject;
import javax.inject.Named;
import org.springframework.transaction.annotation.Transactional;
@Named
public class CustomerServiceImpl implements CustomerService{
    @Inject
    private CustomerDao customerDao;
    @Transactional(propagation=Propagation.REQUIRES_NEW)
    public void save(Customer c){ customerDao.save(c); } }
```

Output

- Customer and Payment entities are successfully saved

- The problem is in the risk of data integrity violation – rollback of PaymentServiceImpl.add transaction does not cause rollback of CustomerServiceImp.save transaction

- See P364AddCustPayment project for the full text

Payment Entity

```
@Entity
public class Payment {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    private java.sql.Date dt;
    . . . . . .
    @ManyToOne
    @JoinColumn(name="customerId")
     private Customer customer;
    . . . . . .
```

Customer Entity

```
@Entity
public class Customer {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private int id;
    . . . . . .
    @OneToMany(mappedBy="customer",
cascade=CascadeType.PERSIST)
    private Collection<Payment> payments;
    . . . . . .
```

Main Class

```
Customer c = new Customer();
. . . . . .
Payment p = new Payment();
. . . . . .
ArrayList<Payment> list = new ArrayList<Payment>();
list.add(p);
c.setPayments(list);
p.setCustomer(c);

customerService.save(c);
```

Output

- Customer and Payment entities are successfully saved
- The problem of integrity violation is solved
- See P365AddCustPayment project for the full text