Hibernate & Spring Data JPA

Beginner to Guru

Spring Data JPA Transactions

Spring Data JPA Transactions

- Spring Data JPA by default supports implicit transactions. Meaning repository methods will create a transaction by default, if there is not an active transaction.
- Spring Data JPA has two types of implicit transactions:
  - Read operations are done in a read only context
  - Updates and deletes are done with the default transactional context
- Use read only with caution, dirty checks are skipped, making more performant
  - If object from read only context is updated and saved, you may encounter issues

Spring Boot Testing Transactions

- Spring Boot by default will create a transaction for your tests and roll it back
- The Spring Data JPA Implicit transactions are NOT used in the test context
  - Implicit transactions are only used outside of a transactional context
- If you have a method under test with one or more repository method calls, you may see different results when run outside of the test context
  - Typically a detached entity error from accessing lazy load properties outside the Hibernate context

Declared with the @Transactional Annotation

- Spring Framework provides an @Transactional annotation in the package "org.springframework.transaction.annotation"
- JEE also provides a @Transactional annotation in the package "javax.transaction"
- Spring will support either option
  - Spring 4.x might have some compatibility issues
- Recommended to use Spring Framework's version of @Transactional
  - More versatile and Spring specific than JEE's @Transactional

Spring's @Transactional Annotation

- Transactional Annotation Attributes:
  - **value** / **transactionManager** - the name of the Transaction Manager to use
  - **label** - String to describe a transaction
  - **Propagation** - The Transaction Propagation Type
  - **Isolation** - Transaction Isolation Level
  - **timeout** - Timeout for Transaction to complete
  - **readOnly** - is read only?

Spring's @Transactional Annotation - Cont

- Transactional Annotation Attributes:
  - **rollbackFor** / **rollbackforClassName** - Exceptions to rollback for
  - **NoRollbackFor** / **noRollbackforClassName** - Exceptions to NOT rollback for

@Transactional - Transaction Manager

- Spring Boot will auto-configure an instance of a Transaction Manager depending on your dependencies
- Spring Framework provides an interface called PlatformTransactionManager
  - Implementations available for JDBC, JTA (JEE), Hibernate, etc
  - Spring Boot auto-configures the appropriate implementation
- Auto-Configured instance named ‘transactionManager’

@Transactional - Transaction Propagation

- **REQUIRED** - (Default) - use existing, or create new transaction
- **SUPPORTS** - Use existing, or execute non-transactionally if none exists
- **MANDATORY** - Support current, throw exception in none exists
- **REQUIRES_NEW** - Create new, suspend current
- **NOT_SUPPORTED** - Execute non-transactionally, suspend current transaction if exists
- **NEVER** - Execute non-transactionally, throw exception if transaction exists
- **NESTED** - Use nested transaction if transaction exists, create if not

@Transactional - Transaction Isolation Level

- **DEFAULT** - (Default) Use level of JDBC connection
- **READ_UNCOMMITTED** - Allows for dirty, no-repeatable reads
- **READ_COMMITTED** - Prevent dirty reads, prevents from reading rows with uncommitted changes
- **REPEATABLE_READ** - Prevent dirty reads and non-repeatable reads
- **SERIALIZABLE** - prevent all dirty reads, similar to REPEATABLE_READ, and performs second read to verify

@Transactional - Transaction Timeout

- Default value is -1, which is to use the underlying implementation
- Spring Boot does not override this
- Unless set specifically at the connection level, defaults to the platform setting
  - For MySQL this is 8 hours

@Transactional - Read Only

- By default the readOnly property is set to false
  - Spring Data JPA for implicate transactions of read methods will set this to true
- Using the readOnly property to true does allow for Hibernate to make some efficiency optimizations
  - This is NOT guaranteed
- DO NOT USE if you expect to update and save entities fetched

@Transactional - RollbackFor / NoRollbackFor

- By default unhandled runtime exceptions will be rollback
- Typically default is fine for most situations
- Can be useful where you wish to rollback a child transaction, but not the whole transaction

Using @Transactional at Repository Level

- Spring Data JPA Repository methods can be overridden and customized at the repository level

```
public interface OrderHeaderRepository extends JpaRepository<OrderHeader, Long> {

    new *
    @Override
    @Transactional(readOnly = false)
    Optional<OrderHeader> findById(Long aLong);
```

Implicit Transactions

```
1 usage
public void doSomething(){
    Customer customer = getCustomerMethod1(); //out of scope
    updateCustomerMethod2(customer); //out of scope
}
1 usage
private Customer getCustomerMethod1() {
    return customerRepository.getById(1l); //Implicit Transaction
}
1 usage
private void updateCustomerMethod2(Customer customer) {
    customer.setCustomerName("new Name"); //Implicit transaction
}
```

Don’t Use Private Methods

```
@Transactional
private Customer getCustomerMethod1() {
}
}
```

Methods annotated with '@Transactional' must be overridable

Make 'Bootstrap.getCustomerMethod1' not private ⌥⇧⏎ More actions... ⌥⏎

```
@Transactional
@NotNull
private Customer getCustomerMethod1()
```

guru.springframework.orderservice.bootstrap.Bootstrap

sdjpa-order-service

Declared Transactions & Scope

```
public void doSomething(){
    Customer customer = getCustomerMethod1();
    updateCustomerMethod2(customer);
}
1 usage
@Transactional // 1st Declared Transaction
public Customer getCustomerMethod1() {
    return customerRepository.getById(1l);
}
1 usage
@Transactional // 2nd Declared Transaction
public void updateCustomerMethod2(Customer customer) {
    customer.setCustomerName("new Name");
    customerRepository.save(customer);
}
```

Inherit Transactions

```java
@Transactional // 1st Declared Transaction
public void doSomething(){
    Customer customer = getCustomerMethod1();
    updateCustomerMethod2(customer);
}
1 usage
@Transactional // uses parent transaction
public Customer getCustomerMethod1() {
    return customerRepository.getById(1l);
}
1 usage
@Transactional // uses parent transaction
public void updateCustomerMethod2(Customer customer) {
    customer.setCustomerName("new Name");
    customerRepository.save(customer);
}
```


```java
@Transactional // 1st Declared Transaction
public void doSomething(){
    Customer customer = getCustomerMethod1();
    updateCustomerMethod2(customer);
}
1 usage
public Customer getCustomerMethod1() {
    return customerRepository.getById(1l);
}
1 usage
public void updateCustomerMethod2(Customer customer) {
    customer.setCustomerName("new Name");
    customerRepository.save(customer);
}
```


Child Transactions

```java
@Transactional // 1st Declared Transaction
public void doSomething(){
    Customer customer = getCustomerMethod1();
    updateCustomerMethod2(customer);
}
1 usage
@Transactional(propagation = Propagation.REQUIRED)
public Customer getCustomerMethod1() { // uses parent transaction
    return customerRepository.getById(1l);
}
1 usage
@Transactional(propagation = Propagation.REQUIRES_NEW) //Creates new child transaction
public void updateCustomerMethod2(Customer customer) {
    customer.setCustomerName("new Name");
    customerRepository.save(customer);
}
```

SPRING FRAMEWORK

GURU