Spring Framework
Module 7 – Transactions

Evgeniy Krivosheev
Vyacheslav Yakovenko
Last update: Feb, 2012

Contents

Spring :: Tx :: ACID

- **Atomicity**: either all actions occur, or nothing occurs;
- **Consistency:** once a transaction has completed (successfully or not), the data is in consistent state;
- **Isolation:** multiple users are allowed to process the same data at the same time;
- **Durability:** once a transaction has completed, its result should be durable;

Spring :: Tx :: Types

- Global: are managed by the application server, using JTA (Java Transaction API);
- Local: are resource-specific;

Spring :: Tx

- Transactions support model used in Spring Framework is applicable to different transaction APIs such as JDBC, Hibernate, JPA, JDO, etc.

- Benefits:
  - Application server is NOT required;
  - Declarative transaction management.

Spring :: Tx API

The key Spring transaction abstraction is defined by interface:

- org.springframework.transaction.PlatformTransactionManager

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

Spring :: Tx API

Often used implementations :

- DataSourceTransactionManager ;
- HibernateTransactionManager ;
- JmsTransactionManager ;
- JmsTransactionManager102 ;
- JpaTransactionManager ;
- OC4JJtaTransactionManager ;
- WebLogicJtaTransactionManager ;
- WebSphereUowTransactionManager ;

Spring :: Tx API

PlatformTransactionManager:

- is an interface, and can thus be easily mocked or stubbed to facilitate application testing;
- it is not tied to JNDI and defined in Spring IoC container ;

Spring :: Tx API

The following interface is used for defining and obtaining properties of a specific transaction:

- org.springframework.transaction.TransactionDefinition

Spring :: Tx API

For declarative transactions, TransactionDefinition is created indirectly using annotations. For example:

```
@Transactional(readOnly = true)
public class DefaultFooService implements FooService {

  public Foo getFoo(String fooName) {
       // do something
  }

  @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
  public void updateFoo(Foo foo) {
       // do something
  }
}
```

Spring :: Tx API

However, to make application work with transactions in a declarative fashion it is necessary to add a declaration in application context:

```
<tx:annotation-driven/>
```

And inject one or more transaction manager beans:

```
<bean id="transactionManager"
  class="org.springframework.jdbc.DataSourceTransactionManager">
```

Spring :: Tx API

TransactionDefinition parameters:

- **Isolation**: the degree of isolation that transaction has;
- **Propagation**: transaction propagation through various methods;
- **Timeout**;
- **Read-only** status;

Spring :: Tx :: Isolation Levels

- ISOLATION_DEFAULT
  - Use isolation level of datastore
- ISOLATION_READ_UNCOMMITTED
  - Reading uncommitted changes in current transaction and concurrent transactions
  - Dirty reads, non-repeatable reads and phantom reads can occur
- ISOLATION_READ_COMMITTED
  - Reading all changes in current transaction and committed changes in concurrent transactions
  - Dirty reads cannot occur
  - Non-repeatable reads and phantom reads can occur

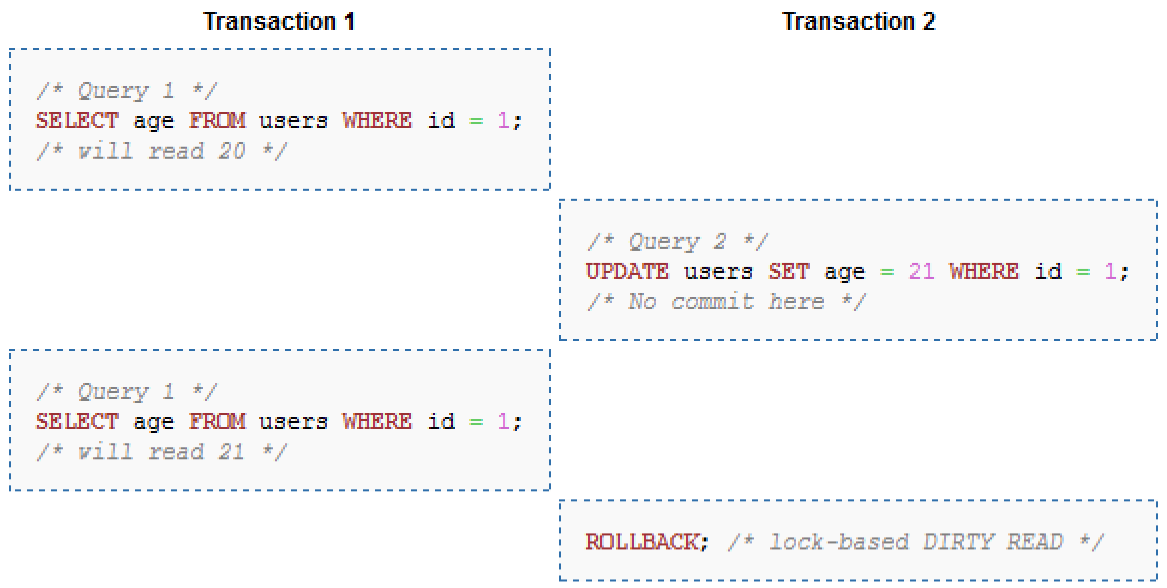
Spring :: Tx :: Isolation Levels

- ISOLATION_REPEATABLE_READ
  - Reading all changes in current transaction. Any changes introduced by concurrent transactions after the current transaction are not available;
  - Dirty reads and non-repeatable reads cannot occur;
  - Phantom reads can occur ;

- ISOLATION_SERIALIZABLE
  - Equal to cases when transactions are executed serially without overlapping;
  - Cannot read all data that was changed from the beginning of transaction, including current transaction;
  - Phantom reads cannot occur;

Spring :: Tx :: Example of Isolation Level Influence

- Dirty reads (Uncommitted Dependency)



Transaction 1

```
/* Query 1 */
SELECT age FROM users WHERE id = 1;
/* will read 20 */
```

Transaction 2

```
/* Query 2 */
UPDATE users SET age = 21 WHERE id = 1;
/* No commit here */
```

Transaction 1

```
/* Query 1 */
SELECT age FROM users WHERE id = 1;
/* will read 21 */
```

Transaction 2

```
ROLLBACK; /* lock-based DIRTY READ */
```

* More information about non-repeatable reads, phantom reads, etc. can be found in training materials

Spring :: Tx :: Isolation Levels

Spring supports following Isolation Levels defined by Enum

`org.springframework.transaction.annotation.Isolation`:

- DEFAULT - Use the default isolation level of the underlying datastore. READ_COMMITTED - A constant indicating that dirty reads are prevented; non-repeatable reads and phantom reads can occur.
- READ_UNCOMMITTED - A constant indicating that dirty reads, non-repeatable reads and phantom reads can occur.
- REPEATABLE_READ - A constant indicating that dirty reads and non-repeatable reads are prevented; phantom reads can occur.
- SERIALIZABLE - A constant indicating that dirty reads, non-repeatable reads and phantom reads are prevented.

Spring :: Tx :: Propagation

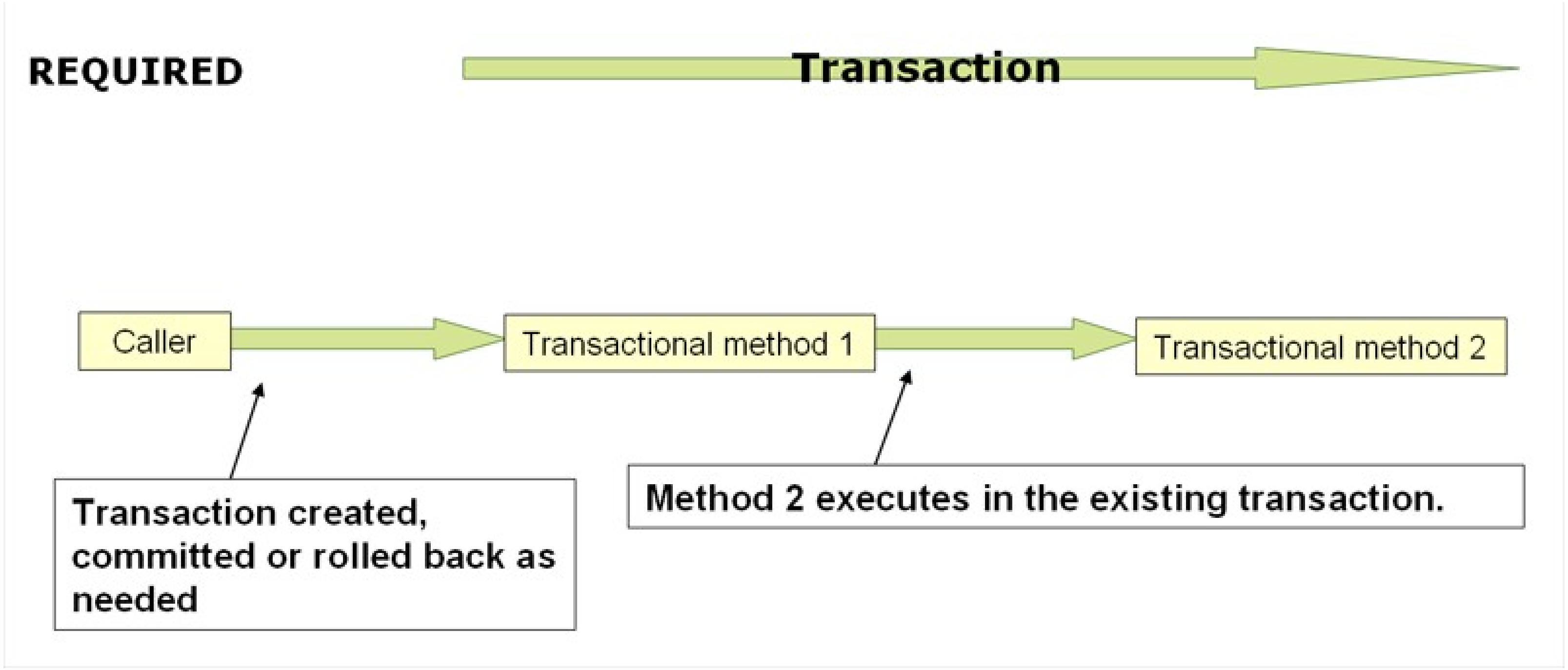
Spring supports following ways of transaction propagation in various methods, as per Enum

`org.springframework.transaction.annotation.Propagation`:

- MANDATORY - Support a current transaction, throw an exception if none exists.
- NESTED - Execute within a nested transaction if a current transaction exists, behave like REQUIRED else.
- NEVER - Execute non-transactionally, throw an exception if a transaction exists.
- NOT_SUPPORTED - Execute non-transactionally, suspend the current transaction if one exists.
- REQUIRED - Support a current transaction, create a new one if none exists.
- REQUIRES_NEW - Create a new transaction, suspend the current transaction if one exists.
- SUPPORTS - Support a current transaction, execute non-transactionally if none exists.

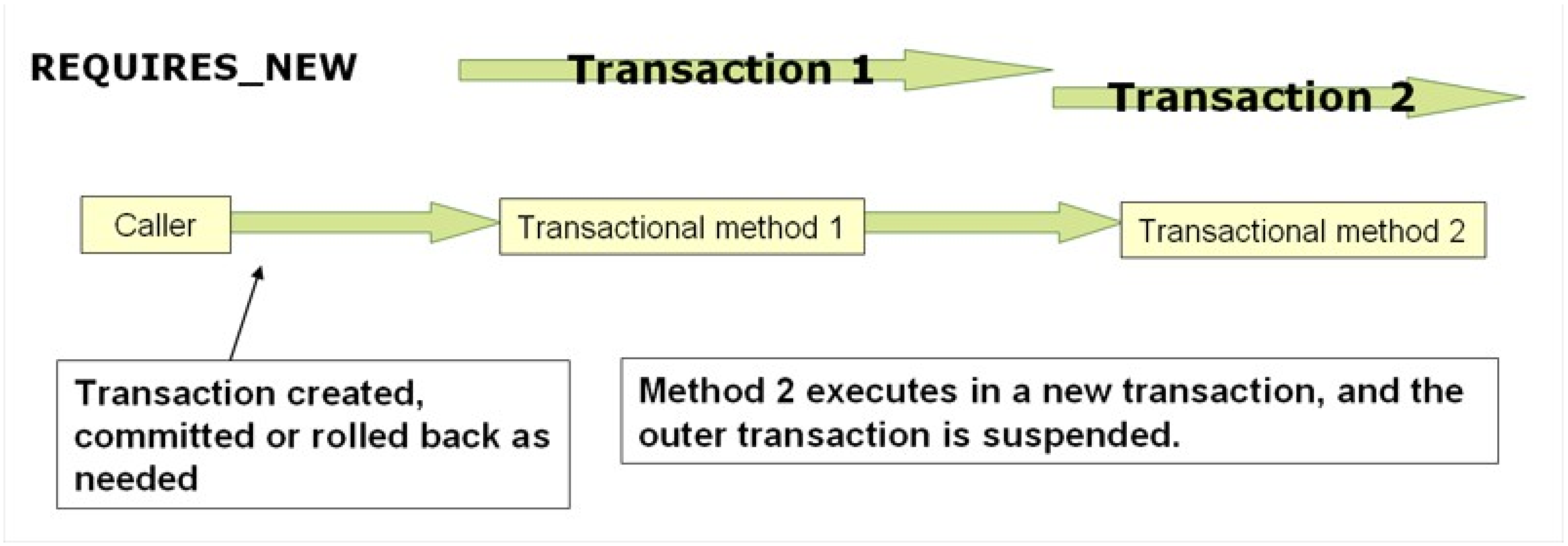
Spring :: Tx :: Propagation

- REQUIRED:


REQUIRED
Transaction
Caller
Transactional method 1
Transactional method 2
Transaction created, committed or rolled back as needed
Method 2 executes in the existing transaction.

Spring :: Tx :: Propagation

- REQUIRES_NEW:


REQUIRES_NEW
Transaction 1
Transaction 2
Caller
Transactional method 1
Transactional method 2
Transaction created, committed or rolled back as needed
Method 2 executes in a new transaction, and the outer transaction is suspended.

Spring :: Tx

- Timeout
  - how long this transaction may run before it is automatically rolled back

- Read-only status
  - a read-only transaction does not modify any data;
  - it can be a useful optimization in some cases.

Spring :: Tx :: Examples

Let’s study some examples of specific transaction management implementations in application context

Spring :: Tx :: Examples

DataSourceTransactionManager:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

   <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>

   <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>

   <property name="username" value="scott"/>

   <property name="password" value="tiger"/>

</bean>


<bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
   <property name="dataSource" ref="dataSource"/>

</bean>
```

Spring :: Tx :: Examples

JtaTransactionManager in J2EE container:

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/jpetstore"  />


<bean id="txManager"
   class="org.springframework.transaction.jta.JtaTransactionManager" />
```

Spring :: Tx :: Rollback

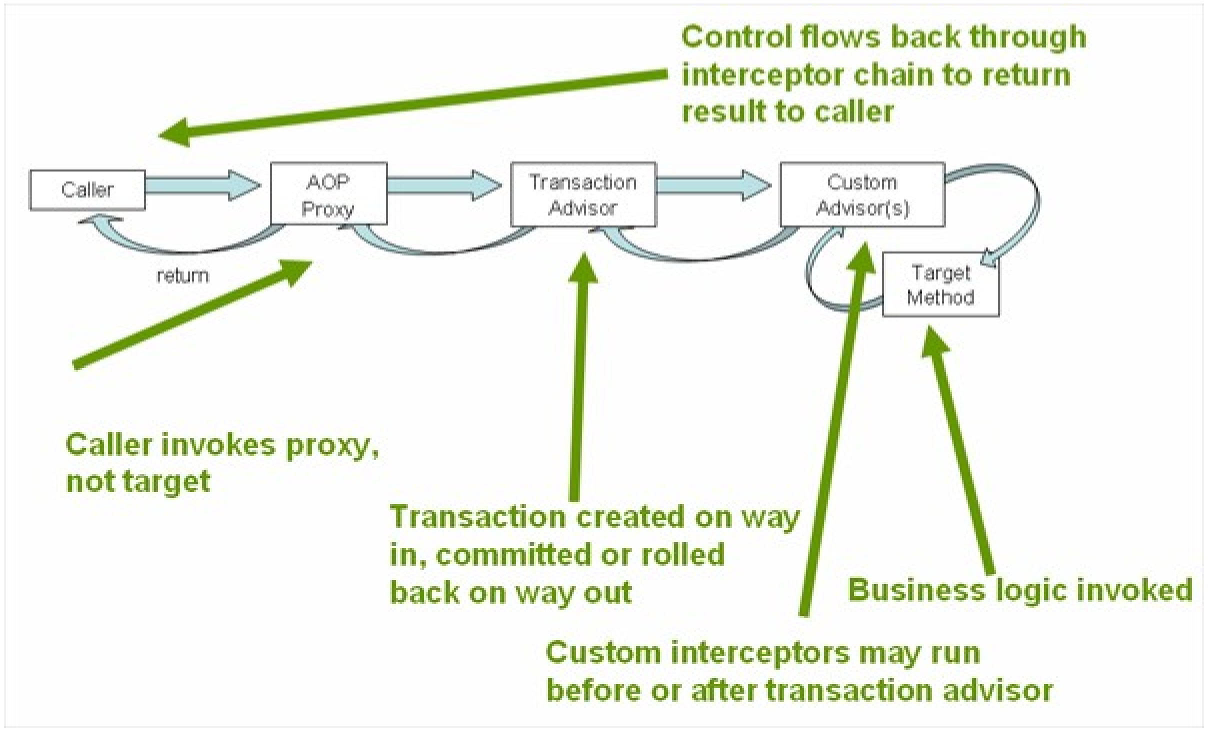
Default Rollback rules:

- They enable us to specify which exceptions should cause automatic roll back;
- By default, transactions are rolled back only with RuntimeException ;
- There are no exceptions for *Exception*;

This behavior can be redefined:

```
@Transactional(rollbackFor = IOException.class,
  noRollbackFor = RuntimeException.class)
public void doSomething() {
  …
}
```

Spring :: Tx + AOP


Control flows back through interceptor chain to return result to caller
Caller
AOP Proxy
Transaction Advisor
Custom Advisor(s)
Target Method
return
Caller invokes proxy, not target
Transaction created on way in, committed or rolled back on way out
Business logic invoked
Custom interceptors may run before or after transaction advisor

Spring :: Tx :: Rollback

@Transactional applied to:

- Interfaces;
- Classes;
- Interface methods;
- Class' public methods;

It is better to apply @Transactional to specific classes and their methods, not to interfaces

Spring :: Tx :: Programmatic management

One possible option is using TransactionTemplate:

```
public class SimpleService implements Service {
    private TransactionTemplate transactionTemplate;
    public Object someServiceMethod() {
        return transactionTemplate.execute(
                new TransactionCallback() {
            public Object doInTransaction(
                    TransactionStatus status) {
                updateOperation1();
                return resultOfUpdateOperation2();
            }
        });
    }
}
```

Spring :: Tx :: Programmatic management

In this case all properties can be defined programmatically:

```
public void nonCallbackService() {
    transactionTemplate.setIsolationLevel(
            TransactionDefinition.ISOLATION_READ_COMMITTED);
    transactionTemplate.setReadOnly(false);
    transactionTemplate.setTimeout(100);
    transactionTemplate.setPropagationBehavior(
            TransactionDefinition.PROPAGATION_REQUIRED);
}
```

Spring :: Tx :: Programmatic management

In this case all properties can be defined programmatically :

- TransactionTemplate supports callback approach;
- Implement TransactionCallback using doInTransaction() method;
- Pass it to execute() method exposed on the TransactionTemplate;

```
public void callbackService() {
    transactionTemplate.execute(new TransactionCallback() {
        public Object doInTransaction(TransactionStatus status){
            updateOperation1();
            return resultOfUpdateOperation2();
        }
    });
}
```

Spring :: Tx :: Programmatic management

- Generally, declarative transaction management is used
- Especially, if there are many transactions in application
- Programmatic management is used in case:
  - There are few transactions in application; TransactionTemplate can be used, but it is not advisable;
  - Transaction name has to be specified explicitly.

Spring :: Tx :: Configuration Example

```
<aop:config>
    <aop:pointcut id="defaultServiceOperation” expression="execution(* x.y.service.*Service.*(..))"/>
    <aop:pointcut id="noTxServiceOperation« expression="execution(*
   x.y.service.ddl.DefaultDdlManager.*(..))"/>
    <aop:advisor pointcut-ref="defaultServiceOperation" advice-ref="defaultTxAdvice"/>
    <aop:advisor pointcut-ref="noTxServiceOperation" advice-ref="noTxAdvice"/>
</aop:config>
<bean id="fooService" class="x.y.service.DefaultFooService"/>
<bean id="anotherFooService" class="x.y.service.ddl.DefaultDdlManager"/>
<tx:advice id="defaultTxAdvice">
    <tx:attributes>
      <tx:method name="get*" read-only="true"/>
      <tx:method name="*"/>
    </tx:attributes>
</tx:advice>
<tx:advice id="noTxAdvice">
    <tx:attributes>
      <tx:method name="*" propagation="NEVER"/>
    </tx:attributes>
</tx:advice>
```

Exercises

№ 8 : Transaction management in Spring

– 30 min for practice;

Any questions!?