Spring Transaction

Spring Framework - Transaction

- Spring Framework has
  - comprehensive transaction support
  - provides a consistent abstraction for transaction management
    - consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate, JPA, and JDO.
    - Supports declarative transaction management.
    - Provides a simpler API for programmatic transaction management than a number of complex transaction APIs such as JTA.
    - Integrates very well with Spring's various data access abstractions

Motivations

- Traditionally, J2EE developers have had two choices for transaction management:
  - *Global* transactions
    - managed by the application server, using the Java Transaction API (JTA)
  - Local transactions
    - are resource-specific
    - the most common example would be a transaction associated with a JDBC connection

Motivations

Global Transactions

- code needs to use JTA, and JTA is a cumbersome API to use (partly due to its exception model)
- a JTA UserTransaction normally needs to be sourced from JNDI
  - we need to use *both* JNDI *and* JTA to use JTA
- use of global transactions limits the reusability of application code, as JTA is normally only available in an application server environment
- EJB CMT uses declarative Transaction Management and removes the need for transaction-related JNDI lookups - but EJB itself necessitates the use of JNDI
  - So to use JTA you need at least EJBs !!!

Motivations

Local Transactions

- easier to use, but have significant disadvantages
  - they cannot work across multiple transactional resources
  - code that manages transactions using a JDBC connection cannot run within a global JTA transaction
  - local transactions tend to be invasive to the programming model

Motivations

- **Spring Transactions**
  - Spring resolves those problems previously discussed
  - It enables application developers to use a *consistent* programming model *in any environment*
    - You write your code once, and it can benefit from different transaction management strategies in different environments

Spring Transaction Model

- The Spring Framework provides both declarative and programmatic transaction management
- Declarative transaction management is preferred by most users, and is recommended in most cases

Programmatic Transaction

- Developers work with the Spring Framework transaction abstraction
- Can run over any underlying transaction infrastructure

Declarative Transaction

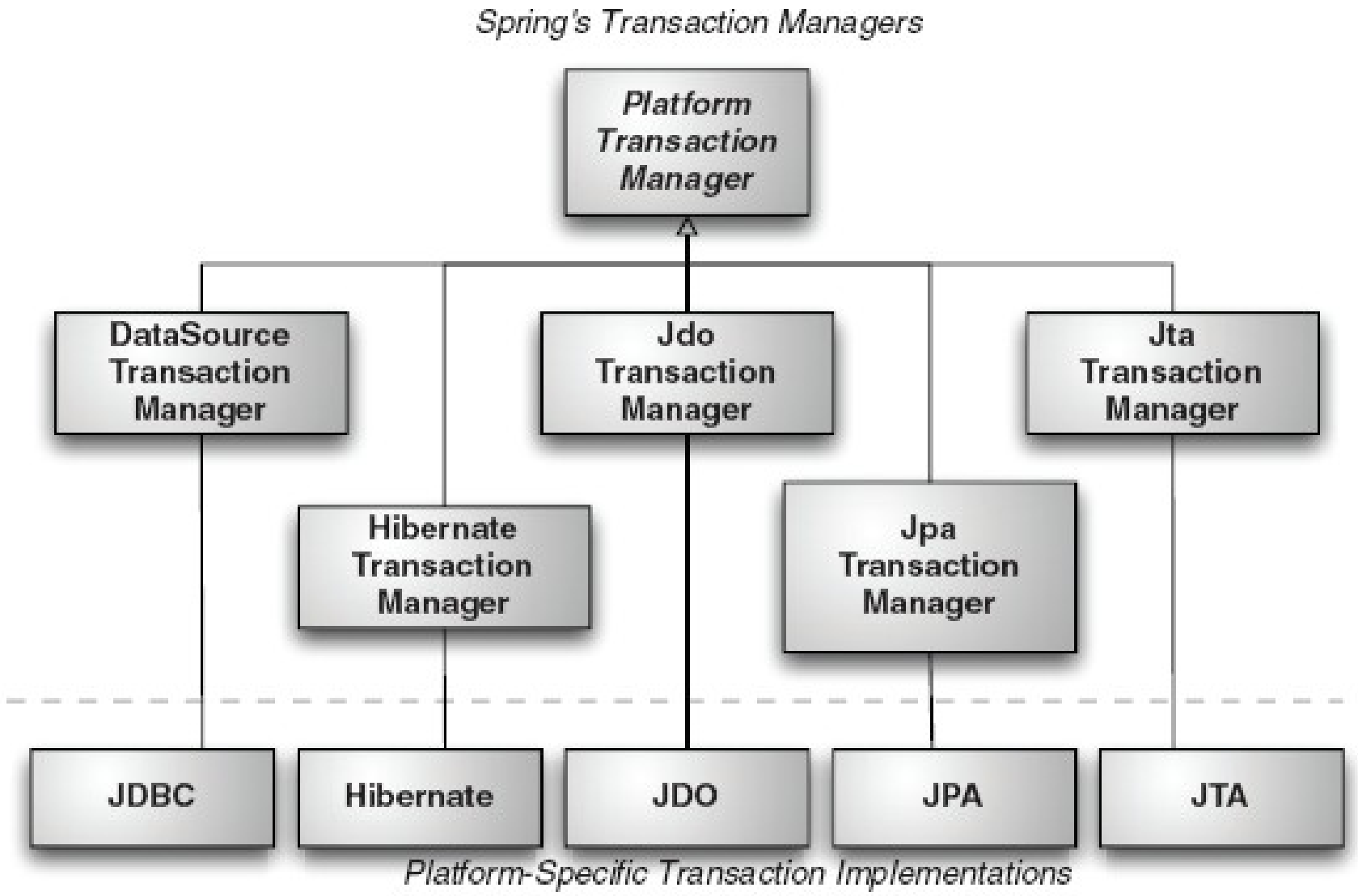
- Developers typically write little or no code related to transaction management
- Don't depend on the Spring Framework's transaction API (or indeed on any other transaction API).

Key abstractions

- The key to the Spring transaction abstraction is the notion of a *transaction strategy*
- A transaction strategy is defined by the org.springframework.transaction.PlatformTransactionManager interface

PlatformTransactionManager

PlatformTransactionManager



PlatformTransactionManager interface

```
public interface PlatformTransactionManager {

    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;

    void rollback(TransactionStatus status) throws TransactionException;
}
```

PlatformTransactionManager interface

- This is primarily an SPI interface, although it can be used programmatically
- PlatformTransactionManager is an *interface*, and can thus be easily mocked or stubbed as necessary
- it is not tied to a lookup strategy such as JNDI
- *PlatformTransactionManager* implementations are defined like any other object (or bean) in the Spring Framework's IoC container

TransactionException

- Can be thrown by any of the PlatformTransactionManager interface's methods
- Is *unchecked* (i.e. it extends the java.lang.RuntimeException class)
  - Transaction infrastructure failures are almost invariably fatal
  - In rare cases where application code can actually recover from a transaction failure, the application developer can still choose to catch and handle TransactionException
  - The salient point is that developers are not *forced* to do so

getTransaction(….) method

- The getTransaction(..) method returns a ***TransactionStatus*** object, depending on a ***TransactionDefinition*** parameter
- The returned ***TransactionStatus*** might represent a new or existing transaction
  - if there were a matching transaction in the current call stack - with the implication being that (as with J2EE transaction contexts) a TransactionStatus is associated with a **thread** of execution).

TransactionDefinition interface

- The TransactionDefinition interface specifies
  - **Isolation**
    - the degree of isolation this transaction has from the work of other transactions. For example, can this transaction see uncommitted writes from other transactions?
  - **Propagation**
    - normally all code executed within a transaction scope will run in that transaction. However, there are several options specifying behavior if a transactional method is executed when a transaction context already exists: for example, simply continue running in the existing transaction (the common case); or suspending the existing transaction and creating a new transaction. *Spring offers all of the transaction propagation options familiar from EJB CMT*.
  - **Timeout**
    - how long this transaction may run before timing out (and automatically being rolled back by the underlying transaction infrastructure).
  - **Read-only status**
    - a read-only transaction does not modify any data. Read-only transactions can be a useful optimization in some cases (such as when using Hibernate).

TransactionStatus interface

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

```
public interface TransactionStatus {
        boolean isNewTransaction();
        void setRollbackOnly();
        boolean isRollbackOnly();
    }
```

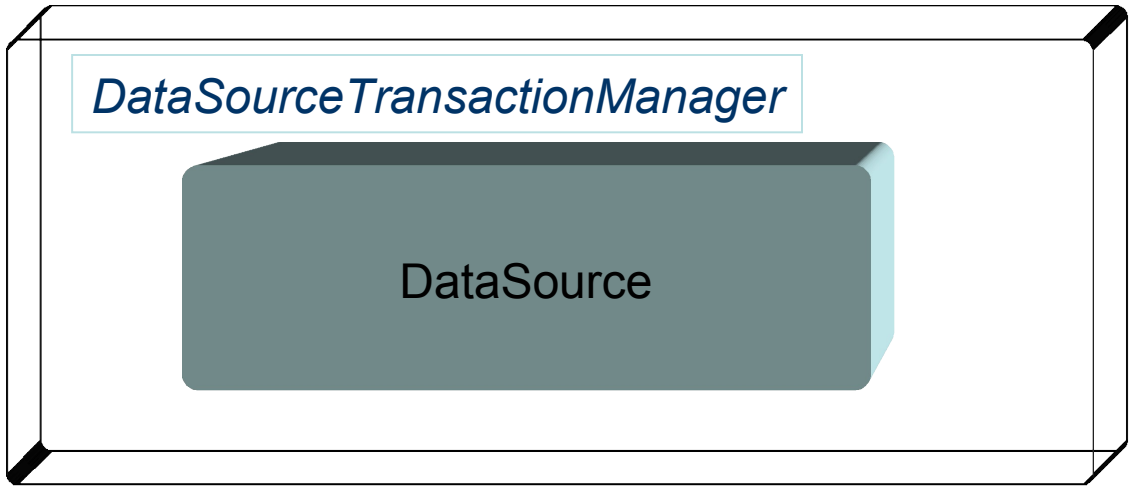
Implementations of PlatformTransactionManager

- Regardless of declarative or programmatic transaction management in Spring, defining the correct PlatformTransactionManager implementation is absolutely essential
- In good Spring fashion, this important definition typically is made using via Dependency Injection
- PlatformTransactionManager implementations normally require knowledge of the environment in which they work: JDBC, JTA, Hibernate, etc.

A local PlatformTransactionManager implementation

(This will work with plain JDBC)

- Define a *JDBC DataSource*, and then use the Spring *DataSourceTransactionManager*, giving it a reference to the *DataSource*



```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
  <property name="driverClassName" value="${jdbc.driverClassName}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</bean>
```

The related PlatformTransactionManager bean definition will look like this

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

- For JTA in a J2EE container, use a container DataSource, obtained via JNDI, in conjunction with Spring's JtaTransactionManager
- The JtaTransactionManager doesn't need to know about the DataSource, or any other specific resources, as it will use the container's global transaction management infrastructure

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee" xsi:schemaLocation="
     http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
     http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">
    <jee:jndi-lookup id="dataSource" jndi-name="jdbc/jpetstore"/>
    <bean id="txManager"
    class="org.springframework.transaction.jta.JtaTransactionManager"/>
    <!-- other <bean/> definitions here -->
</beans>
```

HibernateTransactionManager

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="mappingResources">
    <list>
     <value>org/springframework/samples/petclinic/hibernate/petclinic.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <value>
        hibernate.dialect=${hibernate.dialect}
      </value>
  </property>
</bean>
<bean id="txManager"
      class="org.springframework.orm.hibernate.HibernateTransactionManager">
  <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

Hibernate With JTA transactions

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="mappingResources">
    <list>
      <value>/samples/petclinic/hibernate/petclinic.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <value>
       hibernate.dialect=${hibernate.dialect}
      </value>
  </property>
</bean>
<bean id="txManager"
    class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

Spring Transaction Benefit

- In all cases application code will not need to change at all.
- We can change how transactions are managed merely by changing configuration, even if that change means moving from local to global transactions or vice versa

TransactionAwareDataSourceProxy

Resource synchronization with transactions

Declarative Transaction management

Declarative transaction management

- *Declarative transaction management is Preferred by most users*
- *It is the option with the least impact on application code*
- *most consistent with the ideals of a non-invasive lightweight container*
- Spring's declarative transaction management is made possible with Spring AOP

EJB CMT and Spring Declarative Transaction

- Similarities
  - The basic approach is similar
    - it is possible to specify transaction behavior down to individual method level.
    - It is possible to make a setRollbackOnly() call within a transaction context if necessary

EJB CMT and Spring Declarative Transaction

- Differences
  - EJB CMT is tied to JTA but Spring declarative transaction management works in any environment It can work with JDBC, JDO, Hibernate or other transactions under the covers, with configuration changes only
  - Spring enables declarative transaction management to be applied to any class, not merely special classes such as EJBs
  - Spring offers declarative *rollback rules*: a feature with no EJB equivalent
    - Rollback can be controlled declaratively, not merely programmatically

EJB CMT and Spring Declarative Transaction

- Differences
  - With Spring you can customize Transactional behavior with the help of AOP, With EJB CMT, you have no way to influence the container's transaction management other than setRollbackOnly()
  - Spring does not support propagation of transaction contexts across remote calls, as do high-end application servers. If this feature is needed, the use EJB is recommended
    - However, consider carefully before using such a feature. Normally, we do not want transactions to span remote calls

Declarative Transaction – An Example (1)

```
package com.jp.spring.tx;

public interface FooService {

    Foo getFoo(String fooName);

    Foo getFoo(String fooName, String barName);

    void insertFoo(Foo foo);

    void updateFoo(Foo foo);

}
```

Declarative Transaction – An Example(2)

```
package com.jp.spring.tx;
public class DefaultFooService implements FooService {

    public Foo getFoo(String fooName) {
        throw new UnsupportedOperationException();
    }
    public Foo getFoo(String fooName, String barName) {
        throw new UnsupportedOperationException();
    }

    public void insertFoo(Foo foo) {
        throw new UnsupportedOperationException();
    }

    public void updateFoo(Foo foo) {
        throw new UnsupportedOperationException();
    }
}
```

The applicationContext.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
       http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
       http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">
<bean id="fooService" class="com.jp.spring.tx.DefaultFooService"/>

<tx:advice id="txAdvice" transaction-manager="txManager">
    <!-- the transactional semantics... -->
    <tx:attributes>
      <!-- all methods starting with 'get' are read-only -->
      <tx:method name="get*" read-only="true"/>
<tx:method name="*"/>
    </tx:attributes>
  </tx:advice>
```

```
<aop:config>
  <aop:pointcut id="fooServiceOperation" expression="execution(*
   com.jp.spring.tx.FooService.*(..))"/>
  <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceOperation"/>
</aop:config>
```

```
<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:@localhost:1521:ocl"/>
   <property name="username" value="scott"/>
   <property name="password" value="tiger"/>
</bean>
```

```
<!-- similarly, don't forget the PlatformTransactionManager -->
<bean id="txManager"
   class="org.springframework.jdbc.datasource.DataSourceTransactionManag
   er">
   <property name="dataSource" ref="dataSource"/>
</bean>
</beans>
```

Rolling back

Declaratively Rolling back a Transaction

- Rollback rules enable us to specify which exceptions (and throwables) should cause automatic roll back
- We specify this declaratively, in configuration, not in Java code
- So, while we can still call setRollbackOnly() on the TransactionStatus object to roll the current transaction back programmatically, most often we can specify a rule that MyApplicationException must always result in rollback
- This has the significant advantage that business objects don't need to depend on the transaction infrastructure. For example, they typically don't need to import any Spring APIs, transaction or other

Declaratively Rolling back a Transaction

- Option1(recommended)
  - Throw an application specific exception from the code running in a transactional context
  - Indicate to Spring Framework’s transaction Infrastructure that the transaction to be rolled back whenever above exception is thrown
  - The Spring Framework's transaction infrastructure code will catch any unhandled Exception as it bubbles up the call stack, and will mark the transaction for rollback

Declaratively Rolling back a Transaction

- Option1(recommended)
  - Spring Framework's transaction infrastructure code will, by default, *only* mark a transaction for rollback in the case of runtime, unchecked exceptions
    - that is, when the thrown exception an instance or subclass of RuntimeException. (Errors will also - by default - result in a rollback.)
  - Checked exceptions that are thrown from a transactional method will *not* result in the transaction being rolled back

Declaratively Rolling back a Transaction

- XML configuration that demonstrates how one would configure rollback for a checked, application-specific Exception type

```
<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
     <tx:method name="get*" read-only="false"
                                rollback-for="NoProductInStockException"/>
     <tx:method name="*"/>
  </tx:attributes>
</tx:advice>
```

Programmatically Rolling back a Transaction

- XML configuration that demonstrates how one would configure rollback for a checked, application-specific Exception type

```
public void resolvePosition() {
    try {
        // some business logic...
    } catch (NoProductInStockException ex) {
        // trigger rollback programmatically
  TransactionAspectSupport.currentTransactionStatus()
                                    .setRollbackOnly();
    }
}
```

Configuring different transactional semantics for different beans

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/
      spring-tx-2.0.xsd
    http://www.springframework.org/schema/aop http://www.springframework.org/schema/
      aop/spring-aop-2.0.xsd">
    <aop:config>
        <aop:pointcut id="serviceOperation"
                expression="execution(* x.y.service..*Service.*(..))"/>
        <aop:advisor pointcut-ref="serviceOperation" advice-ref="txAdvice"/>
    </aop:config>
```

```
    <!-- these two beans will be transactional... -->
    <bean id="fooService" class="x.y.service.DefaultFooService"/>
    <bean id="barService" class="x.y.service.extras.SimpleBarService"/>
    <!-- ... and these two beans won't -->
    <bean id="anotherService" class="org.xyz.SomeService"/> <!-- (not in the right
      package) -->
    <bean id="barManager" class="x.y.service.SimpleBarManager"/> <!-- (doesn't end in
      'Service') -->
    <tx:advice id="txAdvice">
        <tx:attributes>
            <tx:method name="get*" read-only="true"/>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>
    <!-- other transaction infrastructure beans such as a PlatformTransactionManager
      omitted... -->
</beans>
```

An example of configuring two distinct beans with totally different transactional settings

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:aop="http://www.springframework.org/schema/aop"
   xmlns:tx="http://www.springframework.org/schema/tx"
   xsi:schemaLocation="
   http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
   http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/
     spring-tx-2.0.xsd
   http://www.springframework.org/schema/aop http://www.springframework.org/schema/
     aop/spring-aop-2.0.xsd">

   <aop:config>
      <aop:pointcut id="defaultServiceOperation"
            expression="execution(* x.y.service.*Service.*(..))"/>
      <aop:advisor pointcut-ref="defaultServiceOperation" advice-ref="defaultTxAdvice"/>

      <aop:pointcut id="noTxServiceOperation"
            expression="execution(* x.y.service.ddl.DefaultDdlManager.*(..))"/>
      <aop:advisor pointcut-ref="noTxServiceOperation" advice-ref="noTxAdvice"/>
   </aop:config>
```

```
  <!-- this bean will be transactional (see the 'defaultServiceOperation' pointcut) -->
 <bean id="fooService" class="x.y.service.DefaultFooService"/>
   <!-- this bean will also be transactional, but with totally different transactional settings --
    >
   <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>
   <!-- other transaction infrastructure beans such as a PlatformTransactionManager
     omitted... -->
</beans>
```

<tx:advice/> settings

- The default <tx:advice/> settings are:
  - The propagation setting is REQUIRED
  - The isolation level is DEFAULT
  - The transaction is read/write
  - The transaction timeout defaults to the default timeout of the underlying transaction system, or none if timeouts are not supported
  - Any RuntimeException will trigger rollback, and any checked Exception will not

<tx:advice/> Tag

<tx:method/> settings

- the various attributes of the <tx:method/> tags that are nested within <tx:advice/> and <tx:attributes/> tags are summarized below

Attribute	Required?	Default	Description
`name`	Yes		The method name(s) with which the transaction attributes are to be associated. The wildcard (*) character can be used to associate the same transaction attribute settings with a number of methods; for example, `'get*'`, `'handle*'`, `'on*Event'`, etc.
`propagation`	No	REQUIRED	The transaction propagation behavior
`isolation`	No	DEFAULT	The transaction isolation level
`timeout`	No	-1	The transaction timeout value (in seconds)
`read-only`	No	false	Is this transaction read-only?
`rollback-for`	No		The `Exception(s)` that will trigger rollback; comma-delimited. For example, `'com.foo.MyBusinessException,ServletException'`
`no-rollback-for`	No		The `Exception(s)` that will *not* trigger rollback; comma-delimited. For example, `'com.foo.MyBusinessException,ServletException'`

Annotation Driven Transaction management

Using *@Transactional*

- The functionality offered by the ***@Transactional*** annotation and the support classes is only available to you if you are using at least Java 5 (Tiger)

Using *@Transactional*

```
<!-- the service class that we want to make transactional
   -->
@Transactional
public class DefaultFooService implements FooService {

    Foo getFoo(String fooName);

    Foo getFoo(String fooName, String barName);

    void insertFoo(Foo foo);

    void updateFoo(Foo foo);
}
```

Using @*Transactional*

- When the above POJO is defined as a bean in a Spring IoC container, the bean instance can be made transactional by adding merely *one* line of XML configuration

```
<!-- this is the service object that we want to make transactional -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>
  <!-- enable the configuration of transactional behavior based on annotations --
     >
  <tx:annotation-driven transaction-manager="txManager"/>
  <!-- a PlatformTransactionManager is still required -->
  <bean id="txManager"
     class="org.springframework.jdbc.datasource.DataSourceTransactionManag
     er">
    <!-- (this dependency is defined somewhere else) -->
    <property name="dataSource" ref="dataSource"/>
  </bean>
    <!-- other <bean/> definitions here -->
</beans>
```

Using @*Transactional*

- The @Transactional annotation may be placed before an interface definition, a method on an interface, a class definition, or a *public* method on a class
- However, the mere presence of the @Transactional annotation is not enough to actually turn on the transactional behavior - the @Transactional annotation *is simply metadata* that can be consumed by something that is @Transactional-aware and that can use the metadata to configure the appropriate beans with transactional behavior
- In the case of the above example, it is the presence of the <tx:annotation-driven/> element that *switches on* the transactional behavior

Using *@Transactional*

- method in the same class takes precedence over the transactional settings defined in the class level annotation.

```
@Transactional(readOnly = true)
public class DefaultFooService implements FooService {
    public Foo getFoo(String fooName) {
        // do something
    }
    // these settings have precedence for this method
    @Transactional(readOnly = false, propagation =
  Propagation.REQUIRES_NEW)
    public void updateFoo(Foo foo) {
        // do something
    }
}
```

@Transactional settings

- The default @Transactional settings
  - The propagation setting is PROPAGATION_REQUIRED
  - The isolation level is ISOLATION_DEFAULT
  - The transaction is read/write
  - The transaction timeout defaults to the default timeout of the underlying transaction system, or none if timeouts are not supported
  - Any RuntimeException will trigger rollback, and any checked Exception will not

Programmatic transaction management

Programmatic Transaction

- Spring provides two means of programmatic transaction management:
  - Using the TransactionTemplate
  - Using a PlatformTransactionManager implementation directly
- The Spring team generally recommend the first approach (i.e. using the TransactionTemplate)
- The second approach is similar to using the JTA UserTransaction API (although exception handling is less cumbersome).

Using the TransactionTemplate

- Adopts the same approach as other Spring *templates* such as JdbcTemplate and HibernateTemplate
- Uses a callback approach
- A TransactionTemplate instance is threadsafe

```
Object result = tt.execute(new TransactionCallback() {


    public Object doInTransaction(TransactionStatus
   status) {
        updateOperation1();
        return resultOfUpdateOperation2();
    }
});
```

Using the *TransactionTemplate*

- If there is no return value, use the convenient TransactionCallbackWithoutResult class via an anonymous class

```
tt.execute(new TransactionCallbackWithoutResult() {
protected void doInTransactionWithoutResult(
                                TransactionStatus status) {
        updateOperation1();
        updateOperation2();
    }
});
```

Code within the callback can roll the transaction back by calling the setRollbackOnly() method on the supplied TransactionStatus object

Using the *TransactionTemplate*

- Application classes wishing to use the TransactionTemplate must have access to a PlatformTransactionManager
  - which will typically be supplied to the class via dependency injection
  - It is easy to unit test such classes with a mock or stub PlatformTransactionManager
    - There is no JNDI lookup or static shenanigans here: it is a simple interface. As usual, you can use Spring to greatly simplify your unit testing

PlatformTransactionManager

- ***PlatformTransactionManager*** can be directly used to manage transaction
  - Simply pass the implementation of the PlatformTransactionManager you're using to your bean via a bean reference
  - Then, using the TransactionDefinition and TransactionStatus objects you can initiate transactions, rollback and commit

PlatformTransactionManager

```
DefaultTransactionDefinition def = new
   DefaultTransactionDefinition();
def.setPropagationBehavior(TransactionDefinition.PROPAGATI
   ON_REQUIRED);

TransactionStatus status = txManager.getTransaction(def);
try {
    // execute your business logic here
}
catch (MyException ex) {
    txManager.rollback(status);
    throw ex;
}
txManager.commit(status);
```

Summary

- Spring Transaction Support
- Different Transaction Managers
- Declarative Transaction Management
  - Using xml based configuration
  - Using @Transactional
- Programmatic Transaction Management
  - Using TransactionTemplate
  - Using PlateformTransactionManager

Thank You!

Resources:

Spring Framework Reference Documentation