



Mindtree

Welcome to possible

Jdbc Template, Hibernate Template and Spring transactions

Objectives

- How to use hibernate & JDBC template

Spring DAO support

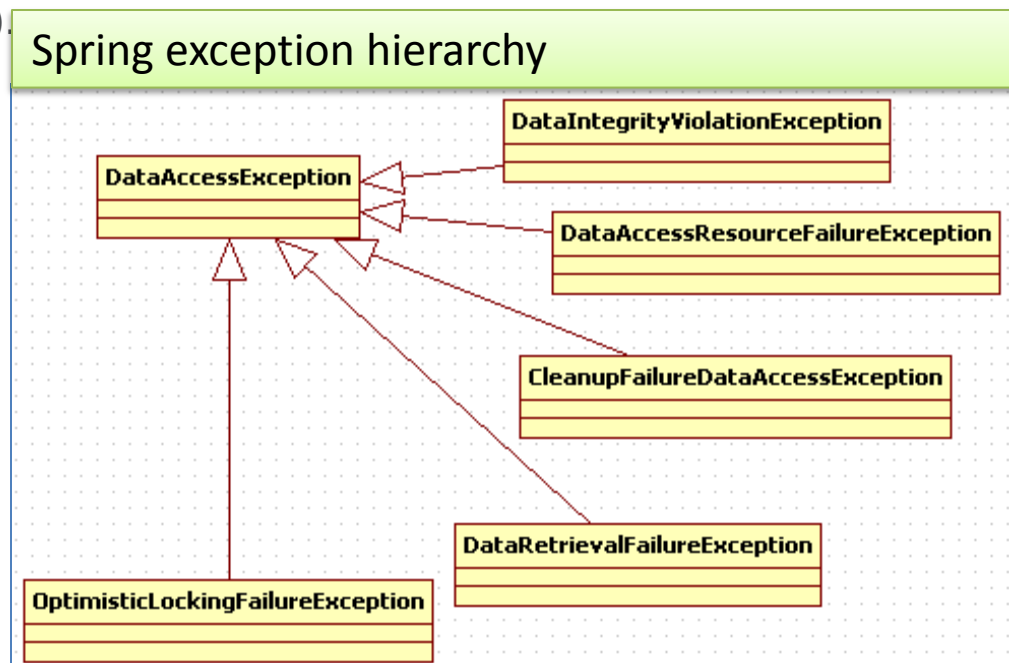
- The Spring Data Access Object (DAO) support is aimed at making it easy to work with data access technologies like JDBC, Hibernate, iBatis, JPA, etc in a consistent way.
- The Spring Data Access Object (DAO) support allows one to switch between the abovementioned persistence technologies fairly easily.

Spring DAO support

- Abstract classes for DAO support
 - JdbcDaoSupport:
 - JdbcDaoSupport class is a superclass for JDBC data access objects.
 - JdbcDaoSupport class requires a DataSource.
 - JdbcDaoSupport class in turn provides a JdbcTemplate instance initialized from the supplied DataSource to subclasses.
 - HibernateDaoSupport:
 - HibernateDaoSupport class is a superclass for Hibernate data access objects.
 - HibernateDaoSupport class requires a SessionFactory.
 - HibernateDaoSupport class in turn provides a HibernateTemplate instance initialized from the supplied SessionFactory to subclasses.
 - HibernateDaoSupport can alternatively be initialized directly via a HibernateTemplate

Spring DAO support

- Exception Hierarchy
 - Spring framework wraps JDBC exceptions, Hibernate exceptions, JPA exceptions, etc into its own exception class hierarchy with `DataAccessException` as the root exception
 - Spring framework converts propriety checked exceptions into a set of focused runtime exceptions (unchecked)



Data access using Spring JDBC support

- Advantage of using Spring Framework's JDBC support.

Action	Spring	Application Developer
Define connection parameters.		YES
Open the connection.	YES	
Specify the SQL statement.		YES
Declare parameters and provide parameter values		YES
Prepare and execute the statement.	YES	
Set up the loop to iterate through the results (if any).	YES	
Do the work for each iteration.		YES
Process any exception.	YES	
Handle transactions.	YES	
Close the connection, statement and resultset.	YES	

Data access using Spring JDBC support

- JdbcTemplate
 - JdbcTemplate simplifies the use of JDBC since it handles the creation and release of resources.
 - JdbcTemplate executes the core JDBC workflow like statement creation and execution, leaving application code to provide SQL and extract results.
 - JdbcTemplate also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the `org.springframework.dao` package.
- Note: DataSource should always be configured as a bean in the Spring IoC container to use JdbcTemplate.

Data access using Spring JDBC support

- context:property-placeholder replaces placeholders \${} with values found in the specified properties file.

jdbc.properties

```
driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/SpringDB
username=root
password=root
```

Spring configuration file

```
<context:property-placeholder
    location="classpath:com/mindtree/dao/jdbc.properties" />

<!-- configure DataSource with properties read from jdbc.properties file -->
<bean id="basicDataSource" destroy-method="close"
    class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="${driver}" />
    <property name="url" value="${url}" />
    <property name="username" value="${username}" />
    <property name="password" value="${password}" />
</bean>
```


Data access using Spring JDBC support

- Dependency injection of DataSource to Dao class

```
<!-- configure DataSource with properties read from jdbc.properties file -->
<bean id="basicDataSource" destroy-method="close"
      class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="${driver}" />
  <property name="url" value="${url}" />
  <property name="username" value="${username}" />
  <property name="password" value="${password}" />
</bean>

<!-- Inject DataSource to AccountDaoJdbcImpl -->
<bean id="accountDao" class="com.mindtree.dao.AccountDaoJdbcImpl">
  <property name="dataSource" ref="basicDataSource"/>
</bean>
```

```
public class AccountDaoJdbcImpl implements AccountDao {
    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        jdbcTemplate = new JdbcTemplate(dataSource);
    }
}
```

Instantiate
JdbcTemplate with
DataSource ref

Data access using Spring JDBC support

- Insert a Row using JdbcTemplate

```
public void createAccount(Account account) throws DaoException {
    String sql = "insert into ACCOUNT(ACCOUNT_NO,ACCOUNT_OWNER,BALANCE) values (?,?,?)";
    Object[] params = new Object[] { account.getAccountNumber(),
        account.getAccountOwner(), account.getBalance() };

    int[] types = new int[] { Types.VARCHAR, Types.VARCHAR, Types.DOUBLE };
    try {
        jdbcTemplate.update(sql, params, types);
    } catch (DataAccessException e) {
        throw new DaoException("Unable to add Account : " + account);
    }
}
```

- Update a Row using JdbcTemplate

```
public void updateAccount(Account account) throws DaoException {
    String sql = "update ACCOUNT set BALANCE = ? where ACCOUNT_NO = ?";
    Object[] params = new Object[] { account.getBalance(),
        account.getAccountNumber() };
    int[] types = new int[] { Types.VARCHAR, Types.VARCHAR, Types.DOUBLE };
    try {
        jdbcTemplate.update(sql, params, types);
    } catch (DataAccessException e) {
        throw new DaoException("Unable to update Account : " + account);
    }
}
```

Data access using Spring JDBC support

- RowMapper: A RowMapper object extracts values from a ResultSet and constructs a entity (domain) object.

```
private static RowMapper accountRowMapper = new RowMapper() {
    @Override
    public Object mapRow(ResultSet rs, int index) throws SQLException {
        Account account = new Account();
        account.setAccountNumber(rs.getString("ACCOUNT_NO"));
        account.setAccountOwner(rs.getString("ACCOUNT_OWNER"));
        account.setBalance(rs.getDouble("BALANCE"));
        return account;
    }
};
```

```
public Account getAccount(String accountNumber) throws DaoException {
    String sql = "select ACCOUNT_NO,ACCOUNT_OWNER,"
        + " BALANCE from ACCOUNT where ACCOUNT_NO = ?";
    try {
        return (Account) jdbcTemplate.queryForObject(sql,
            new Object[] { accountNumber }, accountRowMapper);
    } catch (DataAccessException e) {
        throw new DaoException("Unable to fetch Account : " + accountNumber);
    }
}
```

Data access using Spring JDBC support

- Other important operations using JdbcTemplate:
 - A simple query for getting the number of rows in a relation
 - `int rowCount = jdbcTemplate.queryForInt("select count(0) from ACCOUNT");`
 - Querying for a String
 - `String owner = jdbcTemplate.queryForObject("select ACCOUNT_OWNER from ACCOUNT where ACCOUNT_NO = ?", new Object[] { "SB500"}, String.class);`
 - Invoking a simple stored procedure
 - `jdbcTemplate.update("call ADD_ACCOUNT(?,?,?)",
new Object[] { "SB501", "Banu Prakash", 99993.33});`

Spring and Hibernate Integration

- HibernateTemplate
 - Spring's HibernateTemplate provides an abstract layer over a Hibernate Session.
 - HibernateTemplate's main responsibility is to simplify the work of opening and closing Hibernate Sessions
 - HibernateTemplate also converts Hibernate-specific exceptions to one of the Spring ORM exceptions.

Spring and Hibernate Integration

- LocalSessionFactoryBean
 - Spring's **LocalSessionFactoryBean** is a factory bean that loads one or more Hibernate mapping XML files to produce a Hibernate **SessionFactory**

```
<bean id="mySessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="configLocation">
    <value>classpath:hibernate.cfg.xml</value>
  </property>
</bean>
```

Spring and Hibernate Integration

- Creating HibernateTemplate bean with wired SessionFactory

```
/**
 * @author Banu Prakash © 2011 MindTree Limited
 *
 */
public class AccountDaoHibernateImpl implements AccountDao {

    private HibernateTemplate hibernateTemplate;


    @Autowired
    public void init(SessionFactory sessionFactory) {
        hibernateTemplate = new HibernateTemplate(sessionFactory);
    }

    * (non-Javadoc)
    @Override
    public void createAccount(Account account) throws DaoException {
        try {
            hibernateTemplate.save(account);
        } catch (DataAccessException e) {
            throw new DaoException(e.getMessage(), e);
        }
    }
}
```

Spring and Hibernate Integration

- Using HibernateTemplate

```
public void createAccount(Account account) throws DaoException {
    try {
        hibernateTemplate.save(account);
    } catch (DataAccessException e) {
        throw new DaoException(e.getMessage(), e);
    }
}

* (non-Javadoc) 
@Override
public Account getAccount(String accountNumber) throws DaoException {
    try {
        return (Account) hibernateTemplate.get(Account.class, accountNumber);
    } catch (DataAccessException e) {
        throw new DaoException(e.getMessage(), e);
    }
}
```


Spring and Hibernate Integration

- Using HibernateTemplate

```
* (non-Javadoc)
@SuppressWarnings("unchecked")
@Override
public List<Account> getAllAccounts() throws DaoException {
    try {
        return hibernateTemplate.find("from Account");
    } catch (DataAccessException e) {
        throw new DaoException(e.getMessage(), e);
    }
}

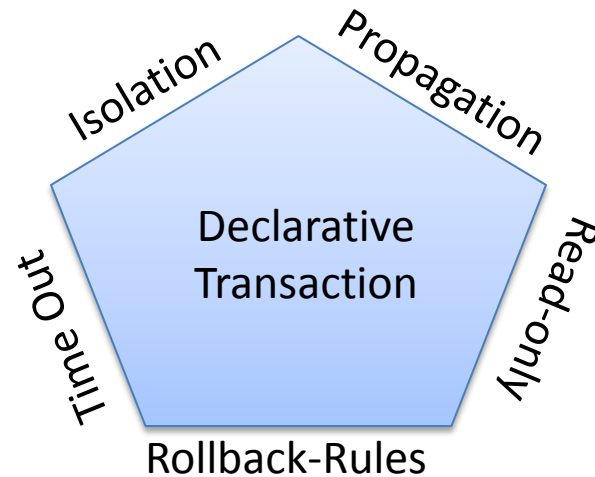
* (non-Javadoc)
@Override
public void updateAccount(Account account) throws DaoException {
    try {
        hibernateTemplate.merge(account);
    } catch (StaleObjectStateException ex) {
        throw new DaoException("Object is modified by other user", ex);
    } catch (DataAccessException e) {
        throw new DaoException(e.getMessage(), e);
    }
}
```

Transaction Management

- The Spring Framework provides a abstraction for transaction management for different transaction APIs such as JTA, JDBC, Hibernate and JPA.
- The Spring Framework supports Declarative transaction management.

Transaction Management

- Declarative Transaction Attributes
 - A transaction attribute is a description of how transaction policies should be applied to a method.
 - There are five attributes to govern how transaction policies are administered.

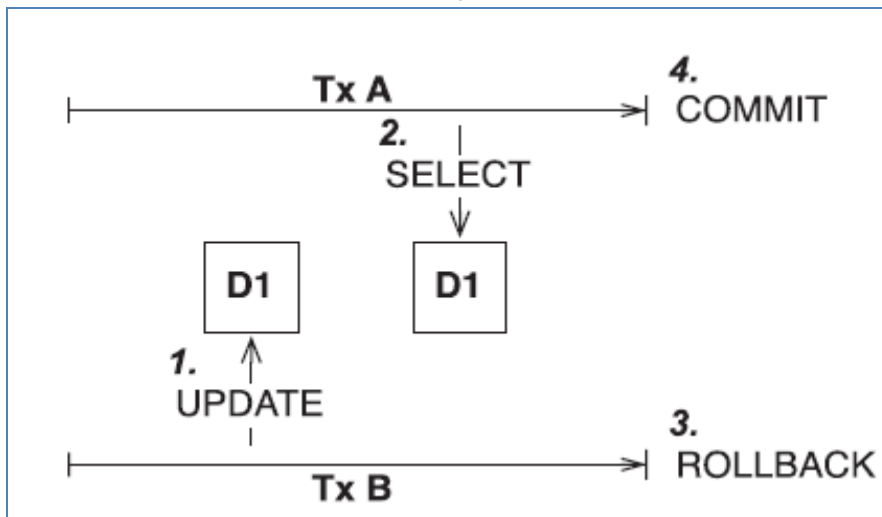


Transaction Management

- Transaction Anomalies
 - Database inconsistencies can occur when more than one transaction is working concurrently on the same objects.
 - In the space of time between when objects are read and then written, the same objects can be read from the database and even manipulated by other transactions. This leads to transaction anomalies
- Different transaction anomalies can be classified as:
 - Dirty Read
 - Non-Repeatable Reads
 - Phantom Reads

Transaction Management

- Transaction Anomalies
 - Dirty Read
 - A dirty read happens when a transaction reads data that is being modified by another transaction that has not yet committed.



TxB begins.

UPDATE employee SET salary = 31650 WHERE empno = '90'

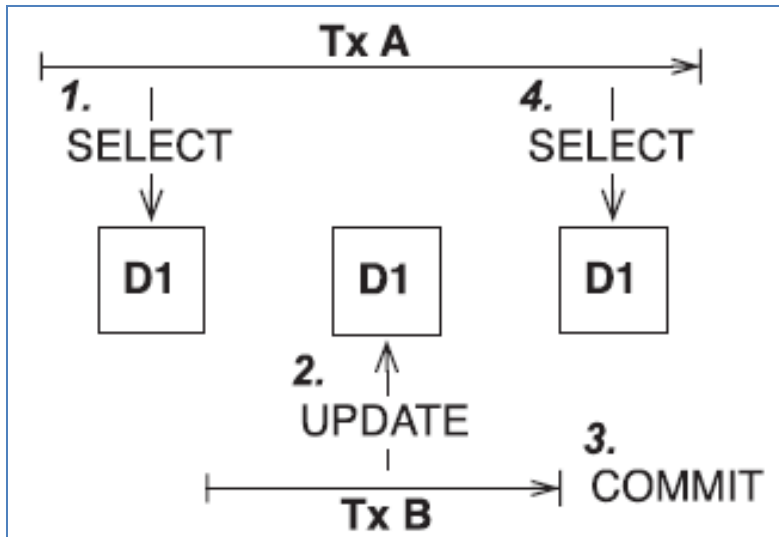
Tx A begins.

SELECT * FROM employee

(Tx A sees data updated by Tx B.
Those updates have not yet been committed.)

Transaction Management

- Transaction Anomalies
 - Non-Repeatable Reads
 - Non-repeatable reads happen when a query returns data that would be different if the query were repeated within the same transaction.
 - Non-repeatable reads can occur when other transactions are modifying data that a transaction is reading



TxA begins.

`SELECT * FROM employee WHERE empno = '90'`

Tx B begins.

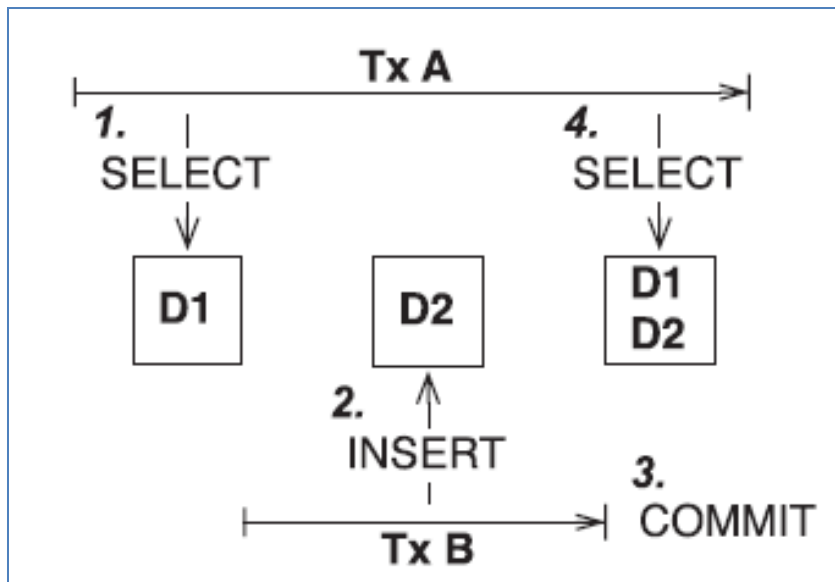
`UPDATE employee SET salary = 30100 WHERE empno = '000090'`

(Tx B updates rows viewed by Tx A before Tx A commits.)

If Tx A issues the same SELECT statement, the results will be different.

Transaction Management

- Transaction Anomalies
 - Phantom Reads
 - Records that appear in a set being read by another transaction.
 - Phantom reads can occur when other transactions insert rows that would satisfy the WHERE clause of another transaction's statement.



Tx A begins.

SELECT * FROM employee WHERE salary > 30000

Tx B begins.

INSERT INTO employee (empno, firstname, salary) VALUES ('390', 'Rahul', 35000)

Tx B inserts a row that would satisfy the query in Tx A if it were issued again

Transaction Management

- Isolation levels and concurrency
 - Setting the transaction isolation level for a connection allows a user to specify how severely the user's transaction should be isolated from other transactions
 - Isolation levels allow you to avoid particular kinds of transaction anomalies.

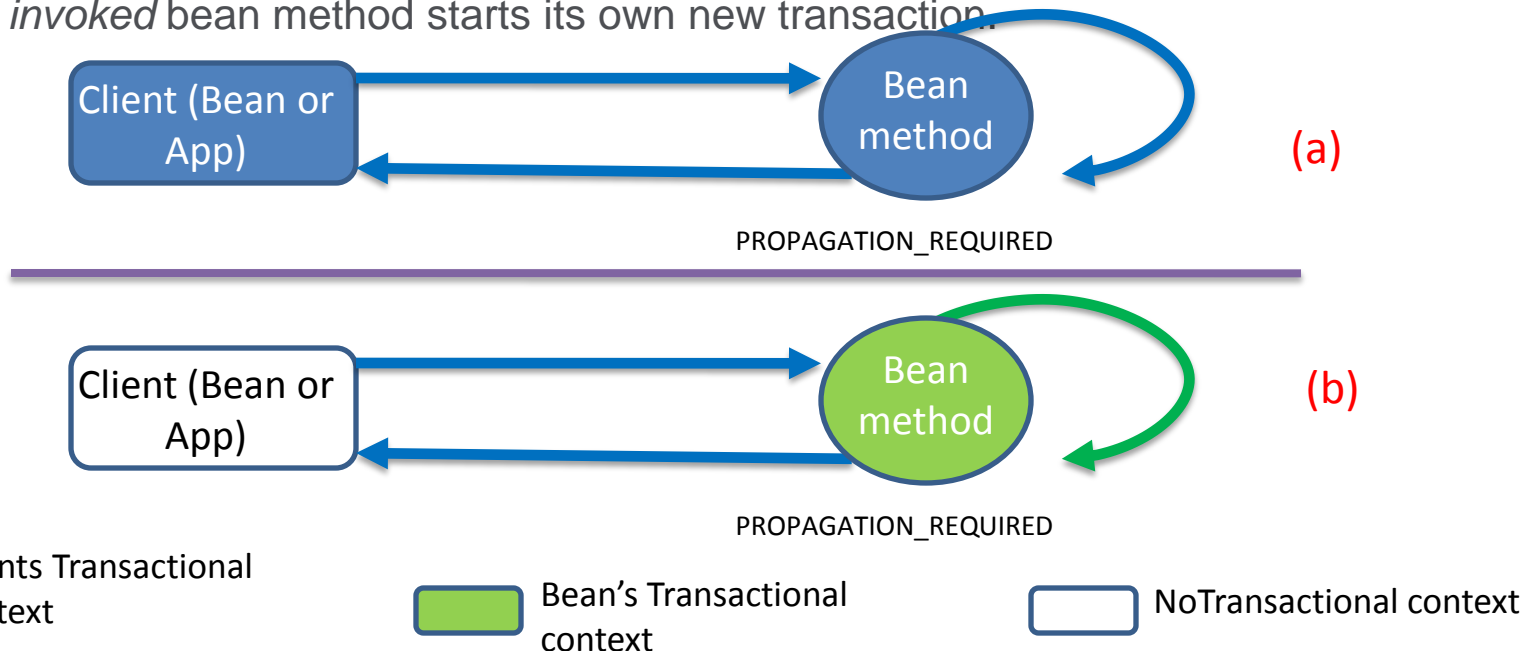
Isolation Level	Dirty Read	Non-Repeatable Reads	Phantom reads
TX_READ_UNCOMMITTED (1)	Possible	Possible	Possible
TX_READ_COMMITTED (2)	Not Possible	Possible	Possible
TX_REPEATABLE_READ (4)	Not Possible	Not Possible	Possible
TX_SERIALIZABLE (8)	Not Possible	Not Possible	Not Possible

Transaction Management

- 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:
 - Continue running in the existing transaction
 - Suspend the existing transaction and create a new transaction.

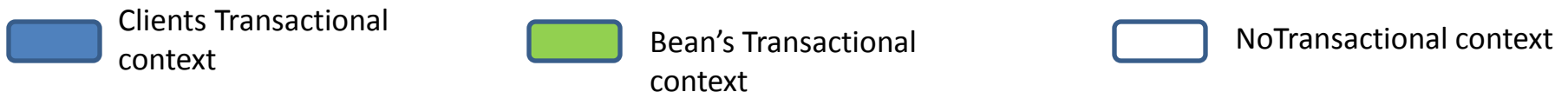
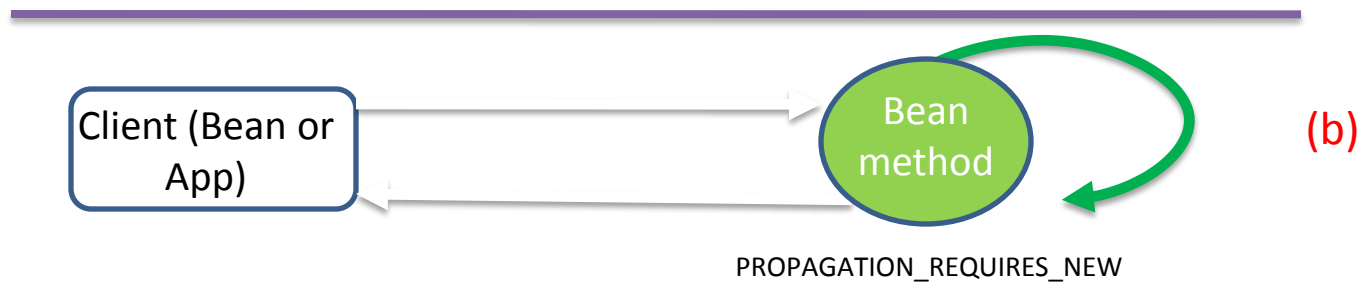
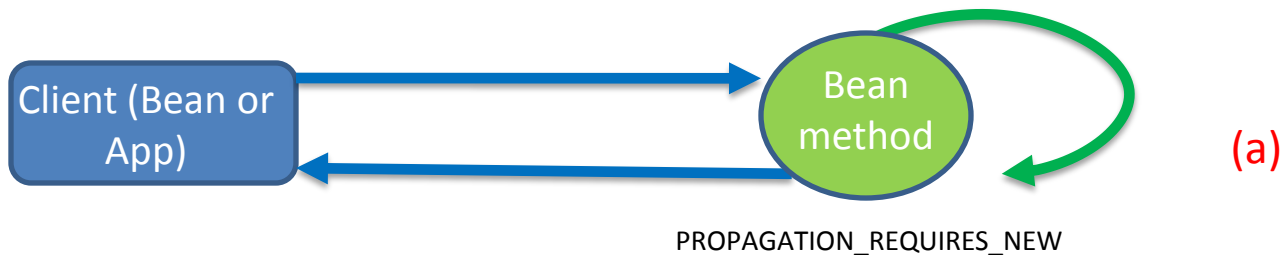
Transaction Management

- Propagation: PROPAGATION_REQUIRED
 - This attribute means that the bean method must be invoked within the scope of a transaction.
 - If the calling client or bean method is part of a transaction, the invoked bean method is automatically included in its transaction scope.
 - If, however, the calling client or bean method is not involved in a transaction, the *invoked* bean method starts its own new transaction.



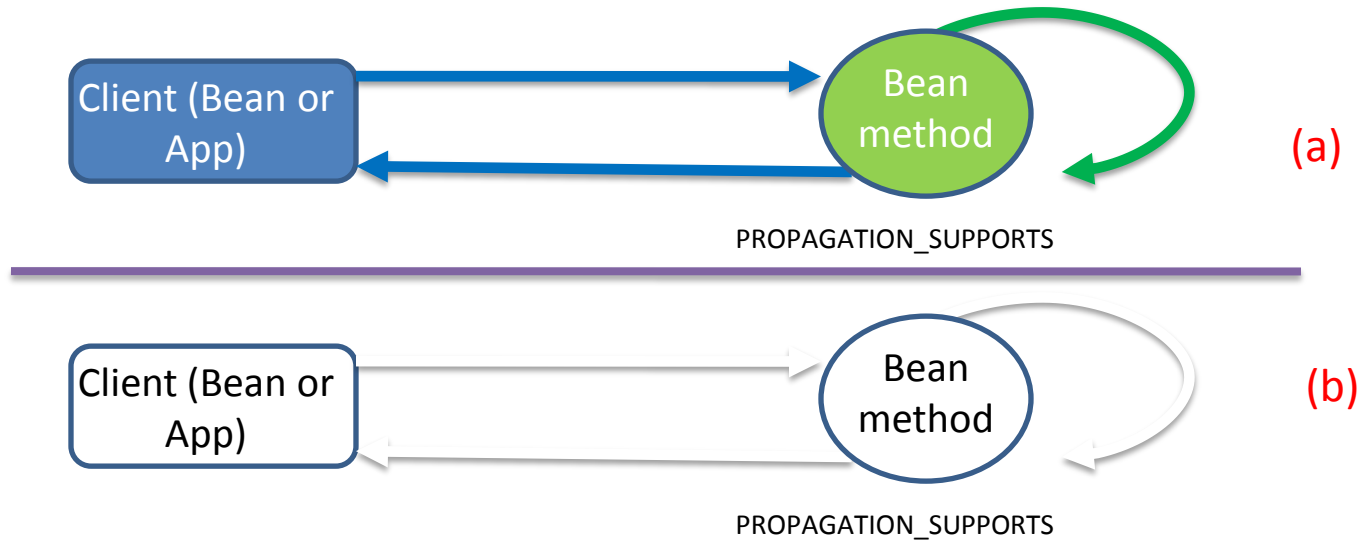
Transaction Management


- Propagation: PROPAGATION_REQUIRES_NEW
 - This attribute means that the when a bean method is invoked transaction is always started.
 - If the calling client or bean method is part of a transaction, that transaction is suspended until the invoked bean's method returns.
 - If, however, the calling client or bean method is not involved in a transaction, the *invoked* bean method starts its own new transaction.





Transaction Management

- Propagation: PROPAGATION_SUPPORTS
 - If the calling client or bean method is part of a transaction, that transaction is propagated to the invoked bean's method.
 - If, however, the calling client or bean method is not involved in a transaction, the *invoked* bean method doesn't have to be part of a transaction .



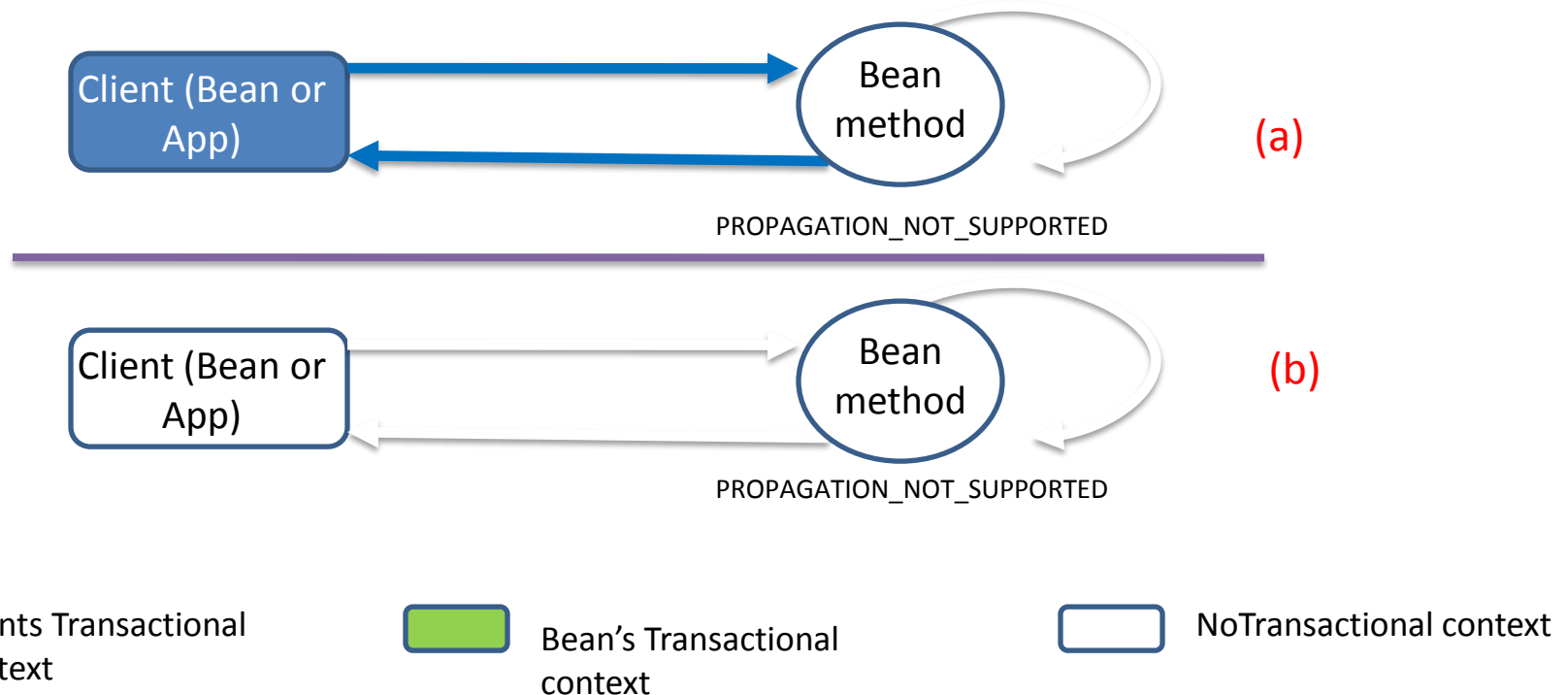
 Clients Transactional context

 Bean's Transactional context

 NoTransactional context

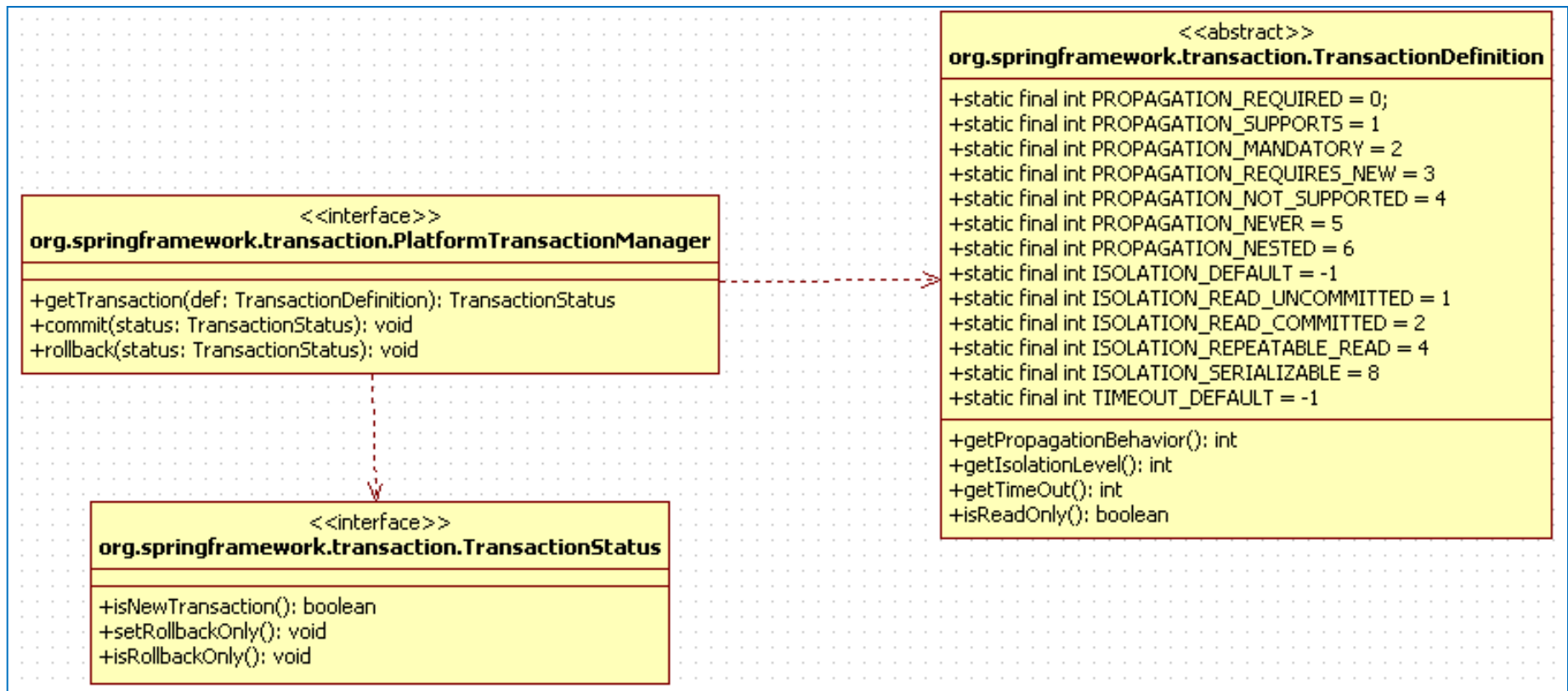
Transaction Management

- Propagation: PROPAGATION_NOT_SUPPORTED
 - Invoking a method on a bean with this transaction attribute suspends the transaction until the method is completed.
 - This means that the transaction scope is not propagated to the *invoked* bean method.
 - Once the invoked bean method is done, the original transaction resumes its execution.



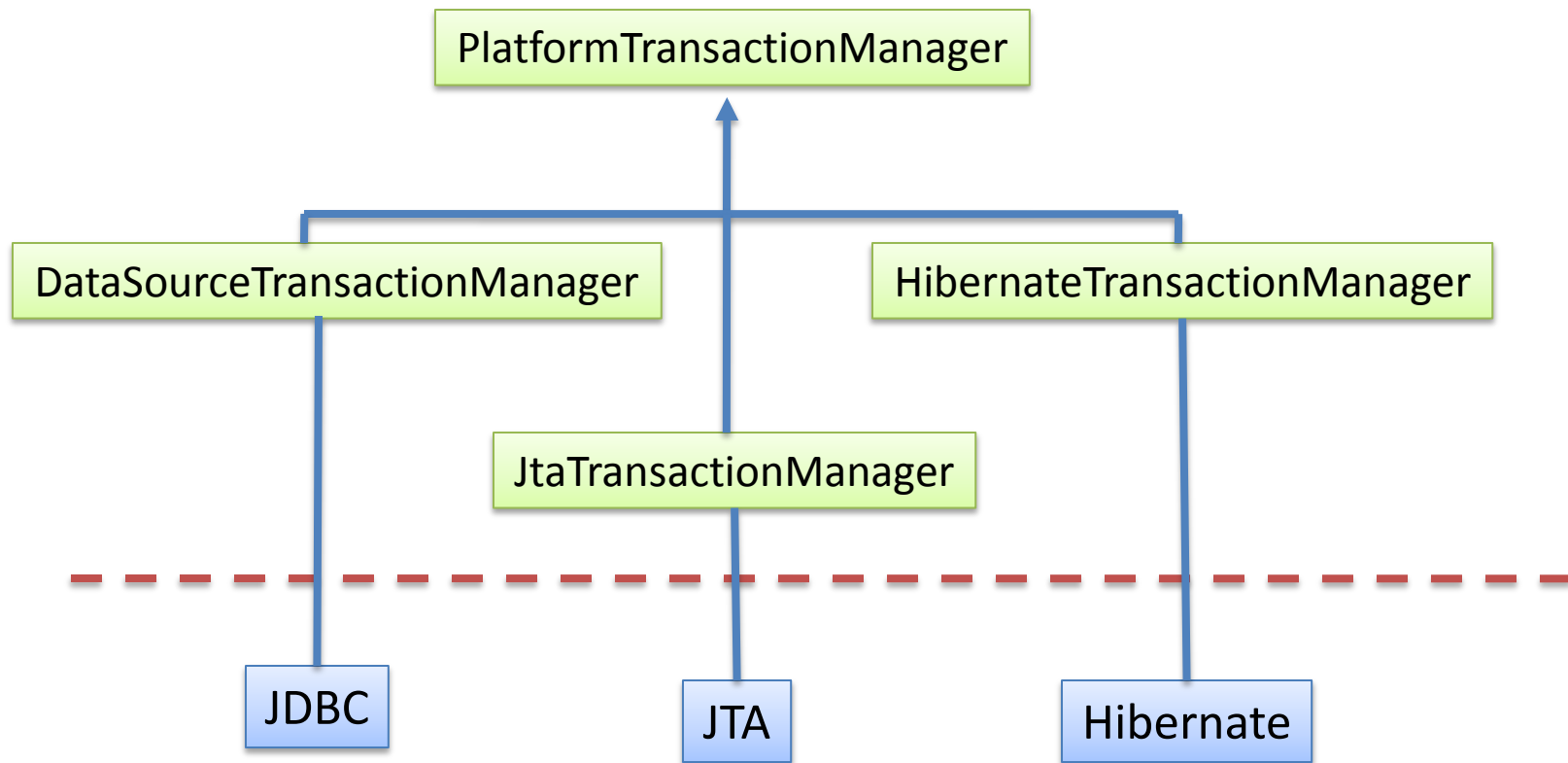
Transaction Management

- Spring Frameworks Transaction API



Transaction Management

- Spring's Transaction Manager's



Transaction Management

- Spring's declarative transaction management.
 - Declarative transaction configuration in versions of Spring 2.0 and above uses <tx:tags /> for transaction declaration.

```
<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"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
</beans>
```


Transaction Management

- Spring's declarative transaction management.
- Configuring declarative transactions

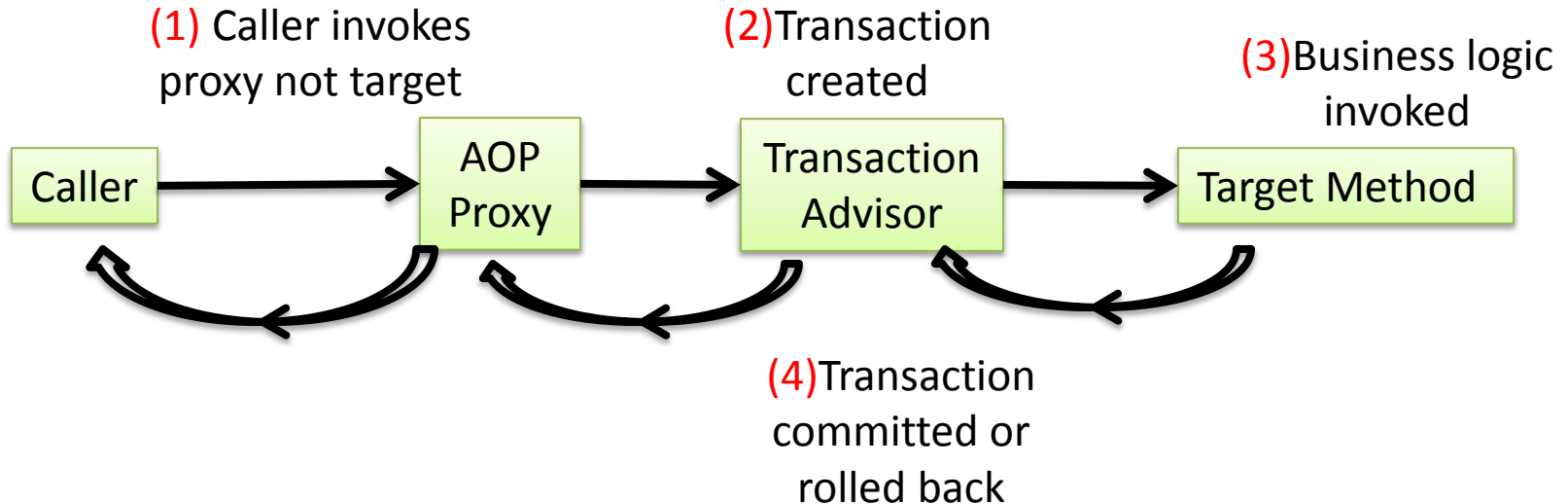
```
<!-- configure PlatformTransactionManager -->
<bean id="transactionManager"
      class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory" />
</bean>

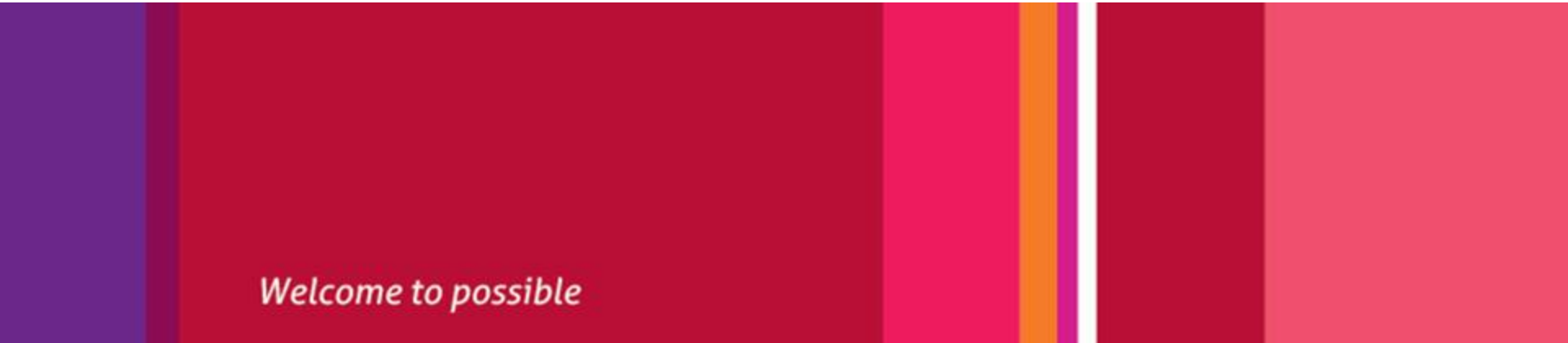
<!-- configure Transaction attributes -->
<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="update*" propagation="REQUIRED" />
        <tx:method name="createAccount" propagation="REQUIRES_NEW" />
        <tx:method name="get*" propagation="SUPPORTS" read-only="true" />
    </tx:attributes>
</tx:advice>

<!-- Apply Transactions using pointcut -->
<aop:config>
    <aop:advisor advice-ref="txAdvice"
        pointcut="execution(* com.mindtree.dao.*(..))" />
</aop:config>
```

Transaction Management

- Applying transaction advice using Spring AOP





Welcome to possible

India | USA | UK | Germany | Sweden | Belgium | France | Switzerland | UAE | Singapore | Australia | Japan | China