Mindtree

Welcome to possible

Spring transactions

Objectives

- Define transaction management
- Understand spring transaction management
- How to use Tx object in spring?
- Understand TX model, Tx propagation

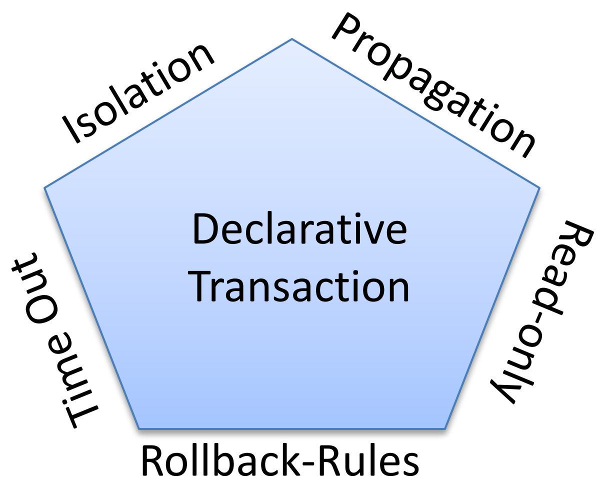
Transaction Management

- 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.



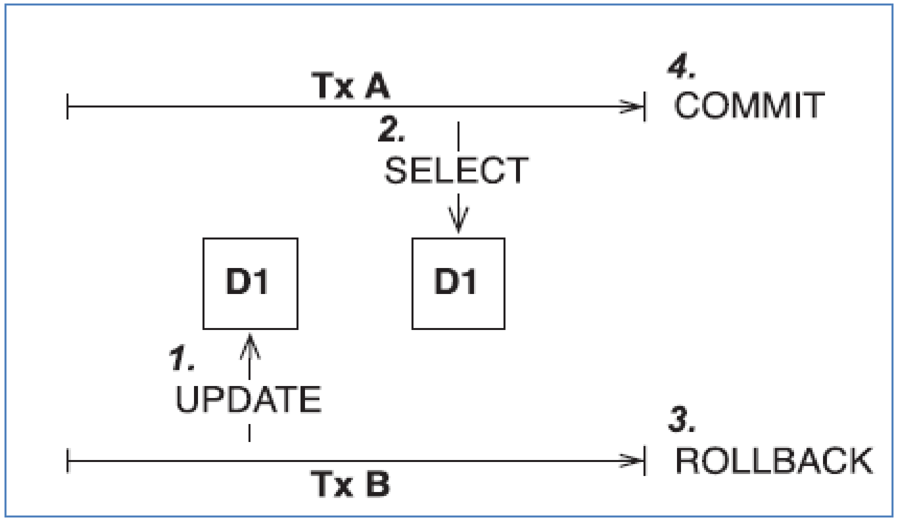
Transaction Management

- 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‘

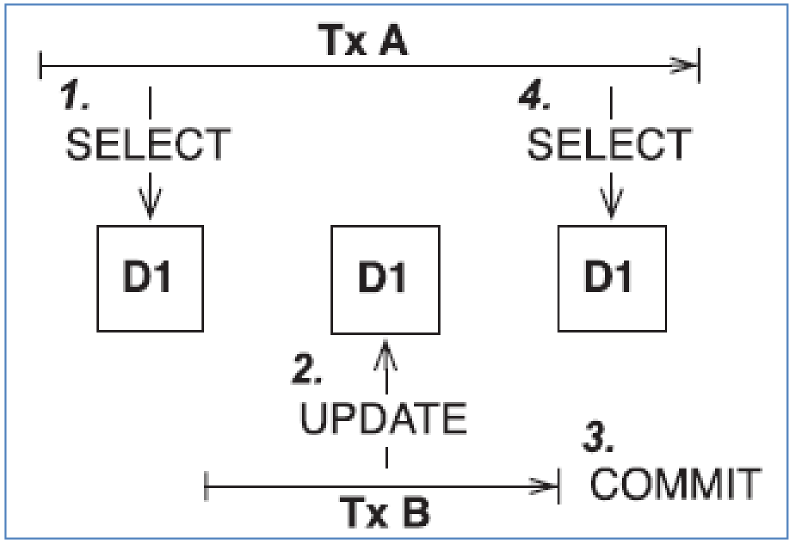
Tx A begins.
SELECT * FROM employee

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

Mindtree

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'

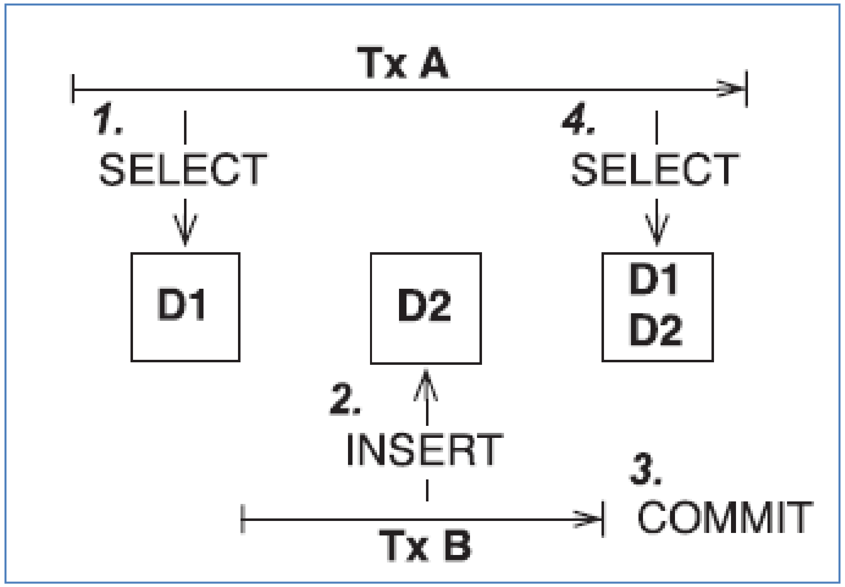
Tx B begins.
UPDATE employee SET salary = 30100 WHERE empno = '000090‘

(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

Mindtree

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.

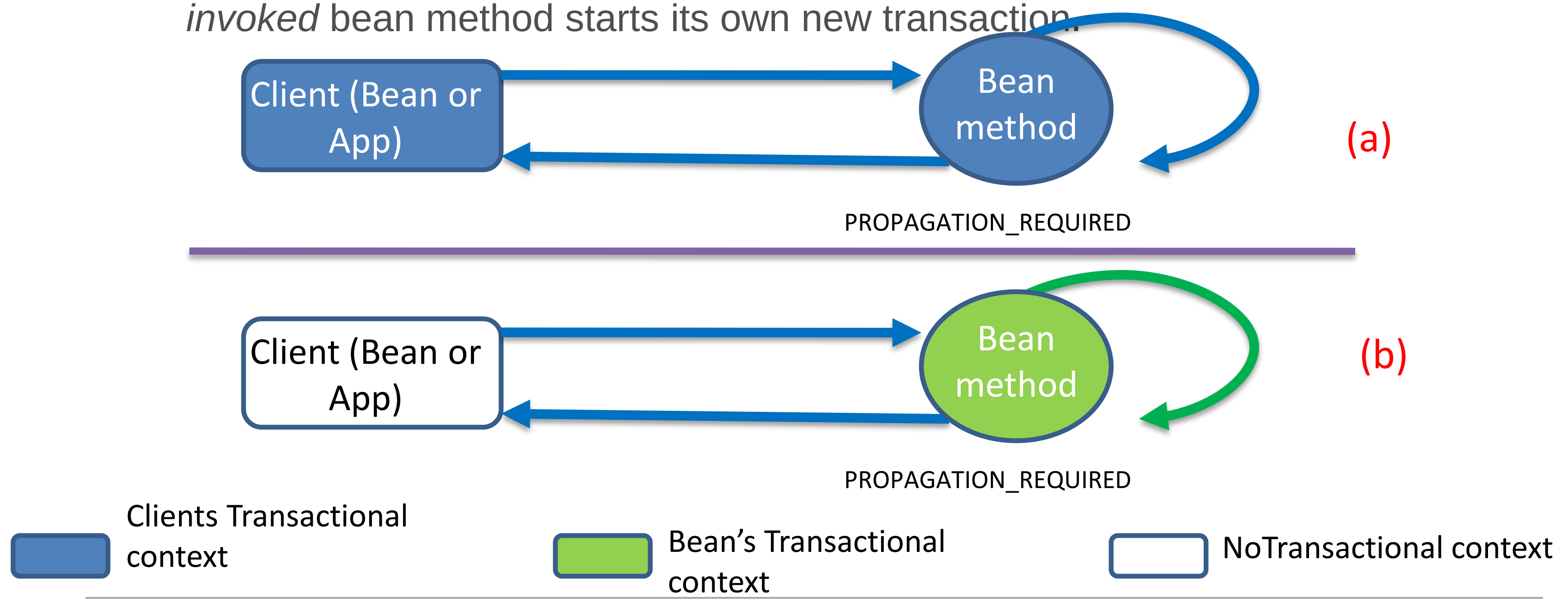
Isolation Level	Dirty Read	Non-Repeatable Reads	Phantom reads
TX_READ_UNCOMMITTED (1)	Possible	Possible	Possible
TX_READ_COMMITTED (2)	Not Possible	Possible	Possible
TX_REPEATABLE_READ (4)	Not Possible	Not Possible	Possible
TX_SERIALIZABLE (8)	Not Possible	Not Possible	Not Possible

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

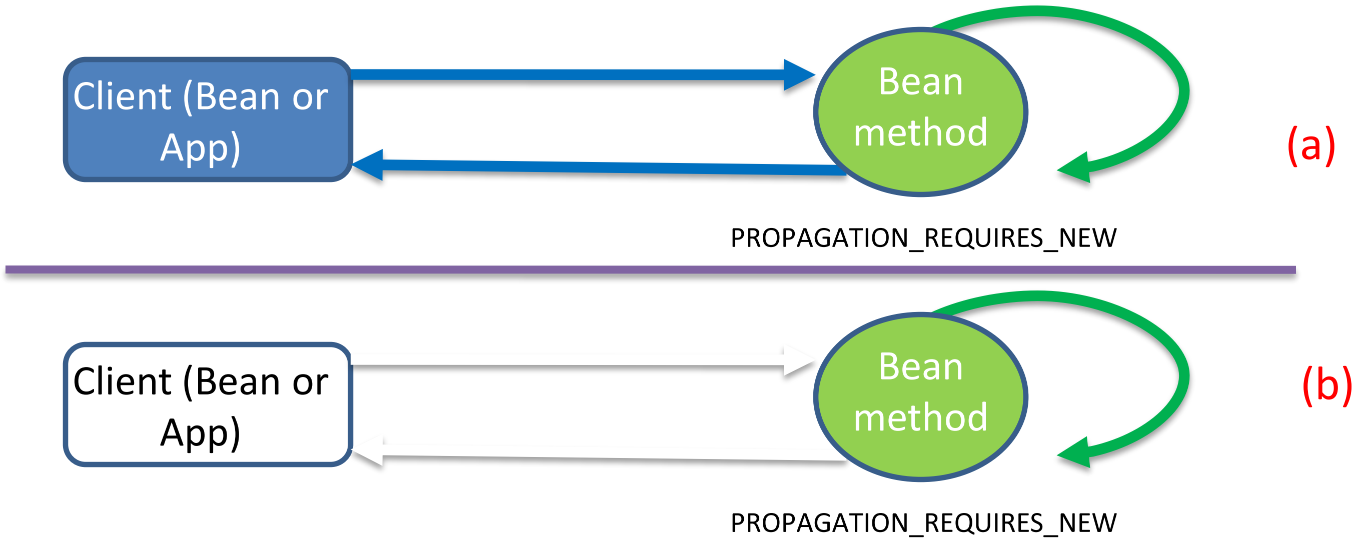
- Propagation: PROPAGATION_REQUIRED
  - This attribute means that the bean method must be invoked within the scope of a transaction.
  - If the calling client or bean method is part of a transaction, the invoked bean method is automatically included in its transaction scope.
  - If, however, the calling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



Mindtree

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.

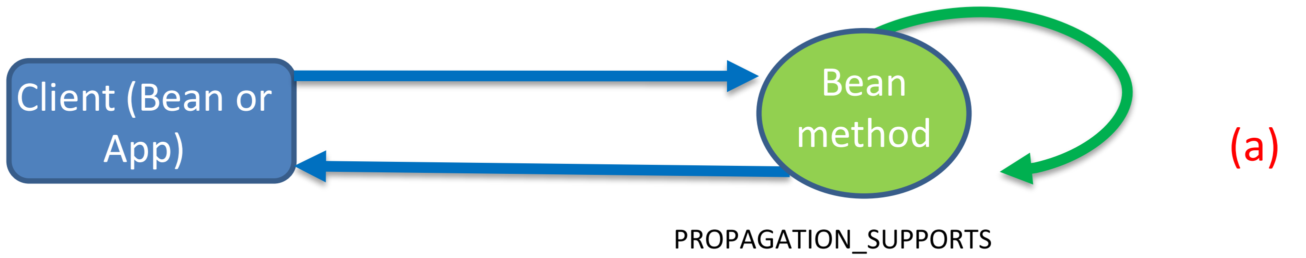


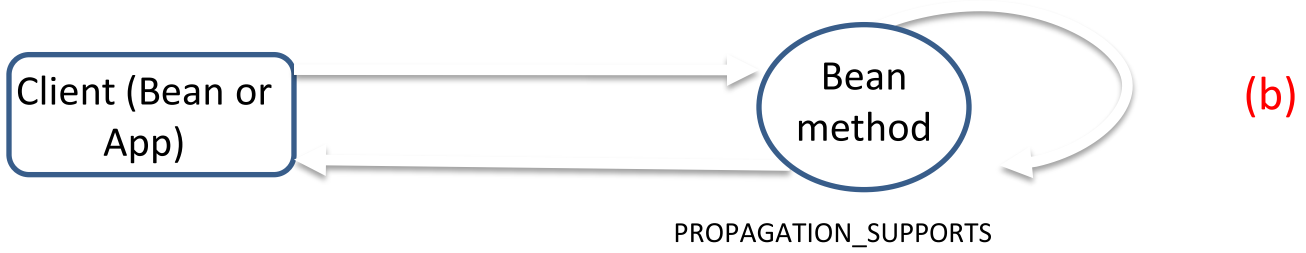
🟦	Clients Transactional context	🟩	Bean's Transactional context	⬜	NoTransactional context

Mindtree

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

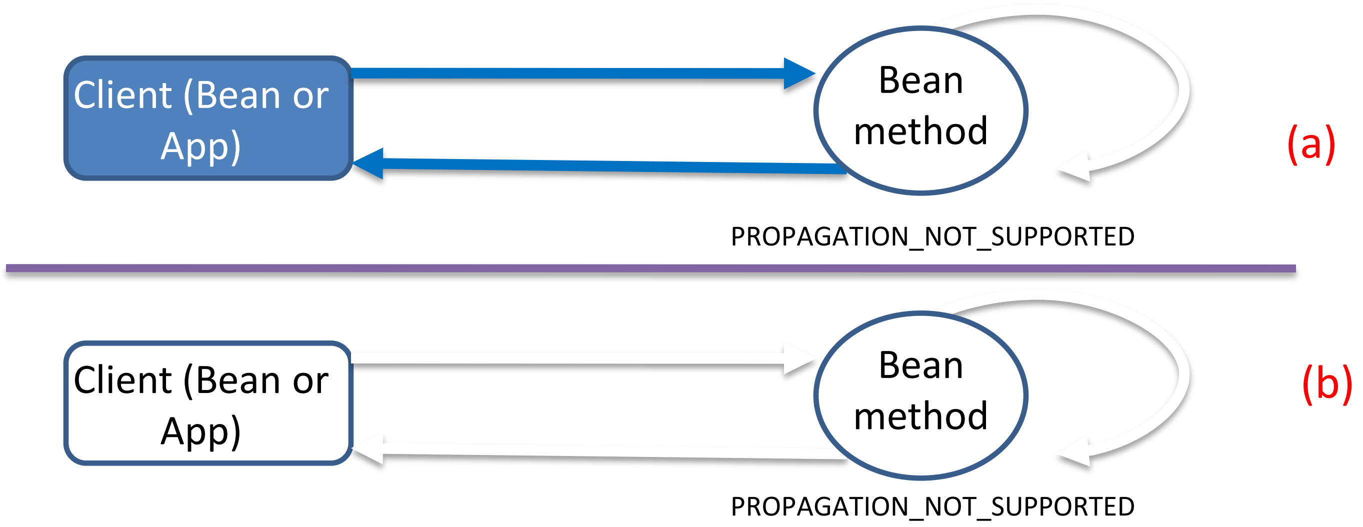
🟩 Bean's Transactional context

⬜ NoTransactional context

Mindtree

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.

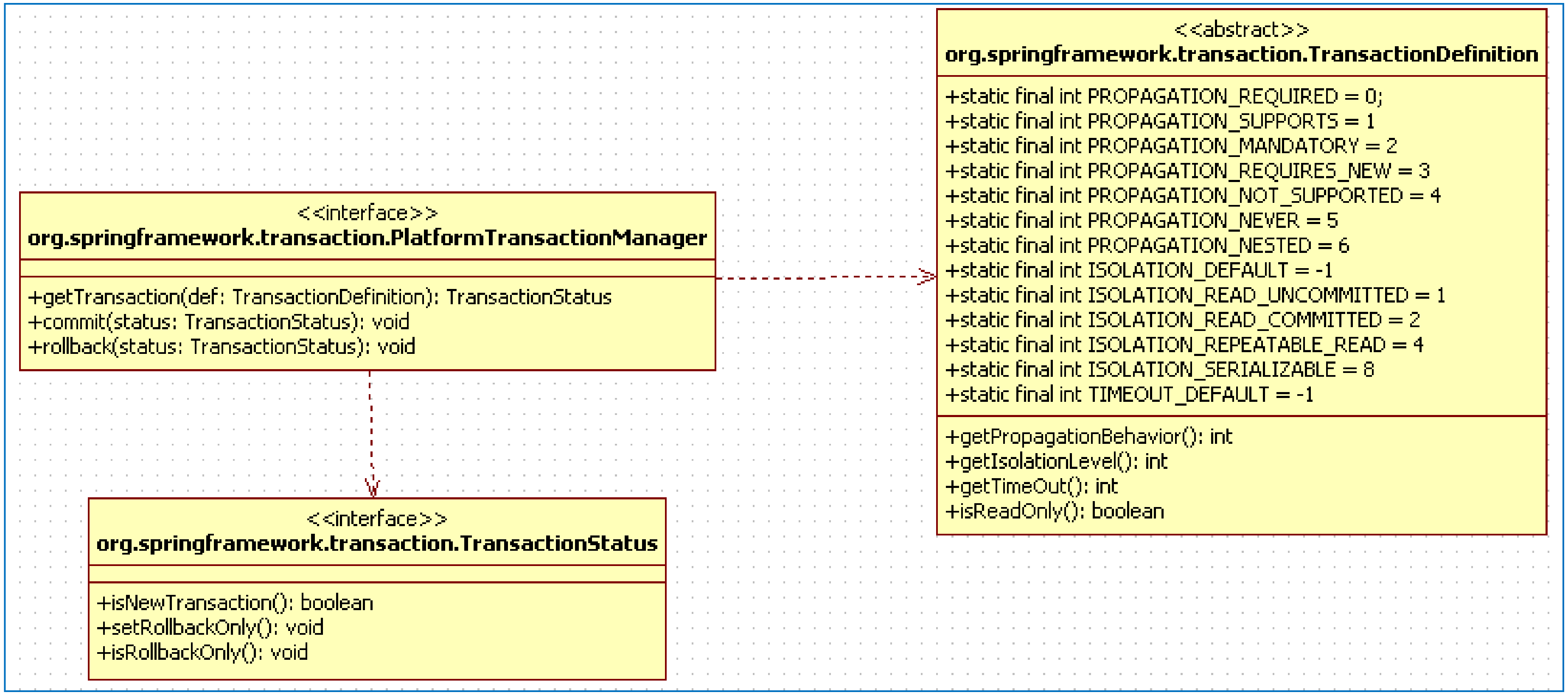


🟦	Clients Transactional context
🟩	Bean's Transactional context
▢	NoTransactional context

Mindtree

Transaction Management

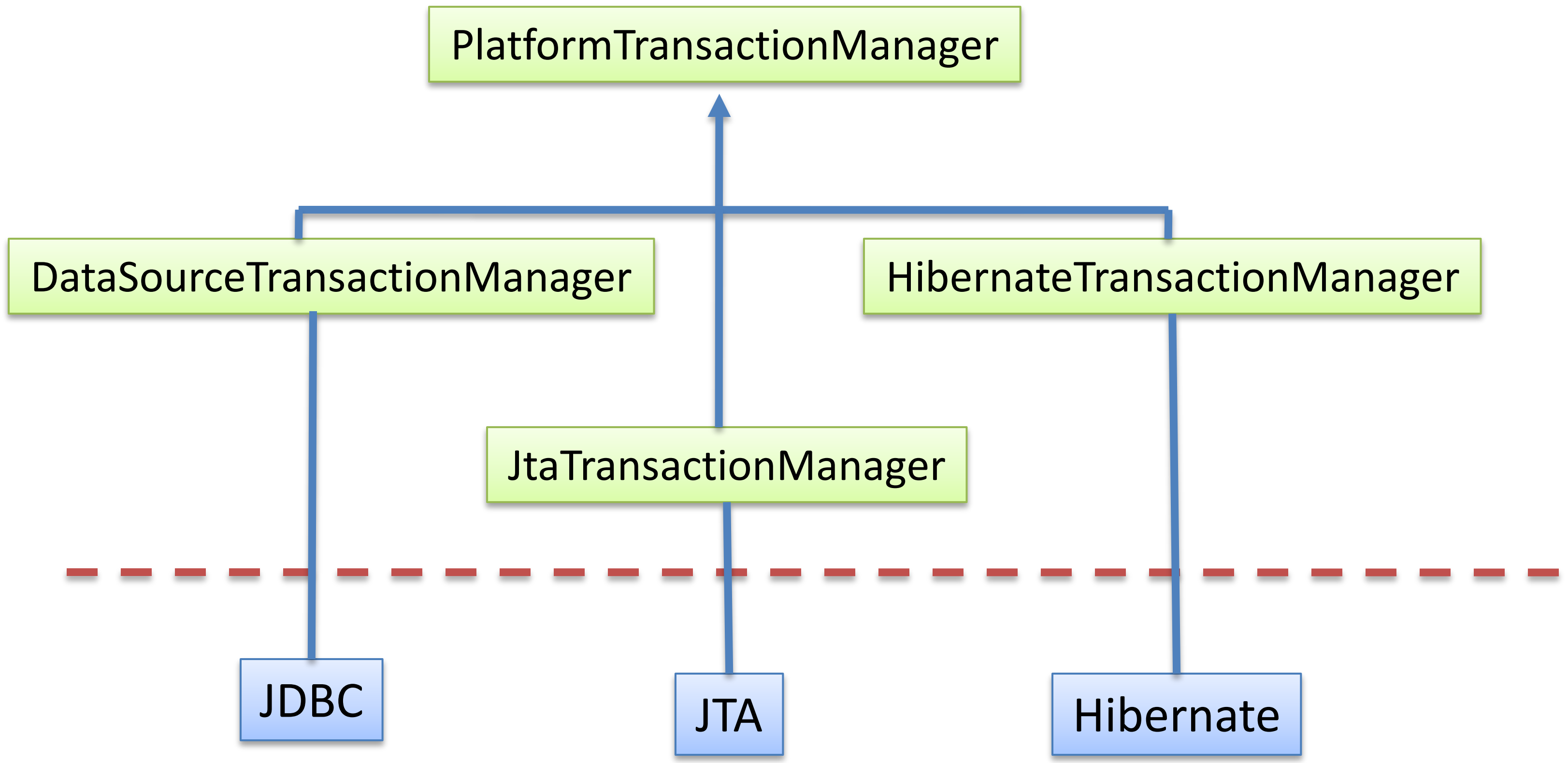
- Spring Frameworks Transaction API



<<interface>>
org.springframework.transaction.PlatformTransactionManager
+getTransaction(def: TransactionDefinition): TransactionStatus
+commit(status: TransactionStatus): void
+rollback(status: TransactionStatus): void
<<interface>>
org.springframework.transaction.TransactionStatus
+isNewTransaction(): boolean
+setRollbackOnly(): void
+isRollbackOnly(): void
<<abstract>>
org.springframework.transaction.TransactionDefinition
+static final int PROPAGATION_REQUIRED = 0;
+static final int PROPAGATION_SUPPORTS = 1
+static final int PROPAGATION_MANDATORY = 2
+static final int PROPAGATION_REQUIRES_NEW = 3
+static final int PROPAGATION_NOT_SUPPORTED = 4
+static final int PROPAGATION_NEVER = 5
+static final int PROPAGATION_NESTED = 6
+static final int ISOLATION_DEFAULT = -1
+static final int ISOLATION_READ_UNCOMMITTED = 1
+static final int ISOLATION_READ_COMMITTED = 2
+static final int ISOLATION_REPEATABLE_READ = 4
+static final int ISOLATION_SERIALIZABLE = 8
+static final int TIMEOUT_DEFAULT = -1
+getPropagationBehavior(): int
+getIsolationLevel(): int
+getTimeOut(): int
+isReadOnly(): boolean

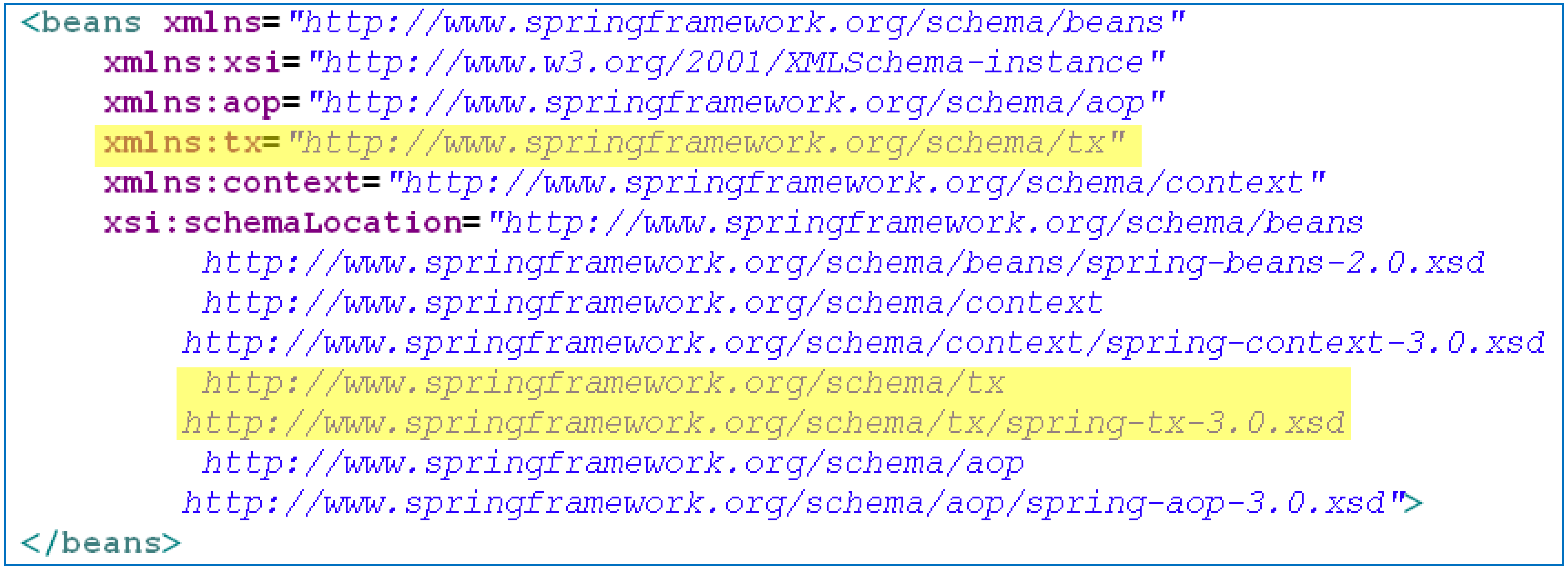
Transaction Management

- Spring's Transaction Manager's


PlatformTransactionManager
DataSourceTransactionManager
HibernateTransactionManager
JtaTransactionManager
JDBC
JTA
Hibernate

Transaction Management

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



```xml
<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

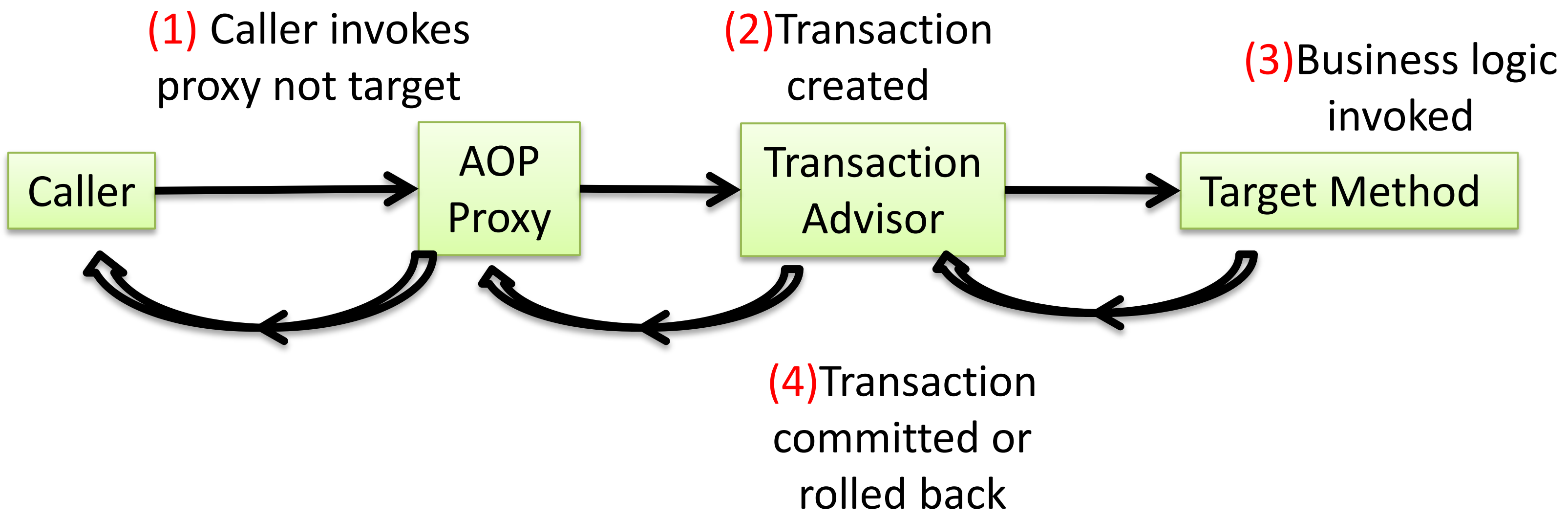
- Spring's declarative transaction management.
  - Configuring declarative transactions

```
<!-- 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>
```

Mindtree

Transaction Management

- Applying transaction advice using Spring AOP


(1) Caller invokes proxy not target
(2)Transaction created
(3)Business logic invoked
Caller
AOP Proxy
Transaction Advisor
Target Method
(4)Transaction committed or rolled back

Mindtree

Welcome to possible

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