

# Spring Transactions



**Stephan Fischli**

# Introduction

- The Spring Framework provides an abstraction for transaction management
  - consistent programming model across different transaction APIs
  - support for declarative transaction management
  - simple API for programmatic transaction management

# Transaction Abstraction

- Spring's transaction abstraction is based on the transaction strategy defined by the `PlatformTransactionManager` interface
- It is primarily a service provider interface (SPI), but can also be used for programmatic transaction management
- Spring Boot auto-configures an implementation depending on the environment in which it is used (JDBC, JPA, JTA, etc.)

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

# The TransactionDefinition Interface

- The `TransactionDefinition` interface specifies the properties of a new transaction to be created or an existing transaction to be retrieved

```
public interface TransactionDefinition {  
    String getName();  
    int getIsolationLevel();  
    int getPropagationBehavior();  
    int getTimeout();  
    boolean isReadOnly()  
};
```

# The TransactionStatus Interface

- The `TransactionStatus` interface allows to control transaction execution and to query transaction status

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

# Local vs Global Transactions

- Local transactions
  - are resource-specific (e.g. transactions using a JDBC connection)
  - are easy to use
- Global transactions
  - allow work with multiple transactional resources (e.g. relational databases and message queues)
  - need a transaction manager to be coordinated
  - are managed through the Java Transaction API (JTA)

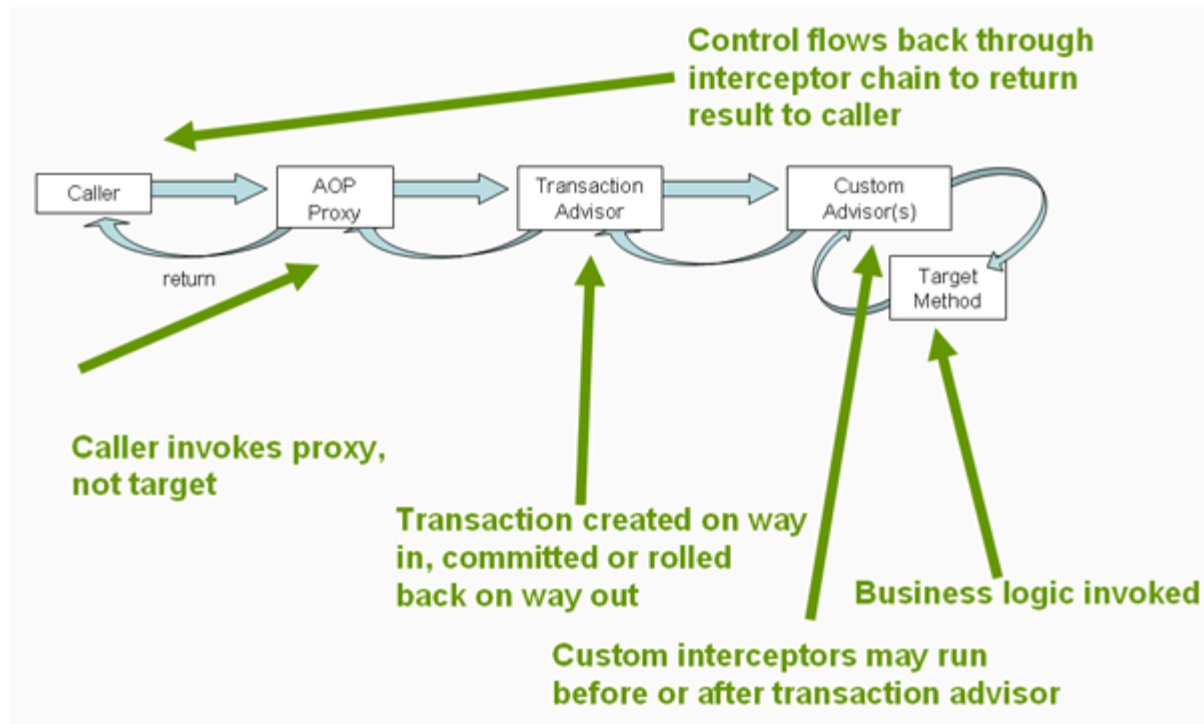
# Transaction Management

# Declarative Transaction Management

- Spring's declarative transaction management
  - works in any transactional environment
  - is applicable to components using annotations
  - offers declarative rollback rules (based on exceptions)

# Transaction Interceptor

- Declarative transaction support is enabled via AOP proxies
- A transaction annotation on a method yields a proxy that uses a transaction interceptor to control the transactions around the method invocations
- The transaction interceptor delegates the transaction handling to the transaction manager



# Transaction Rollback

- The recommended way to roll back a transaction is to throw an exception
- Spring catches any unhandled exception and marks a transaction for rollback in case of an unchecked exception
- Checked exceptions do not result in a rollback by default
- A transaction can also programmatically be marked for rollback using the `TransactionAspectSupport` class

```
public void businessMethod() {  
    ...  
    if (failure) {  
        TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();  
    }  
}
```

# The Transactional Annotation

- The `@Transactional` annotation causes methods to be executed within a transaction
- It can be used at the class level or at the method level which takes precedence
- To enable annotation-driven transaction management, a configuration class may need to be annotated with `@EnableTransactionManagement`

```
@Service
@Transactional
public BusinessService {

    @Transactional(...)
    public void businessMethod() {
        ...
    }
}
```

# Transactional Mode

- Only external method calls that pass through the AOP proxy of a bean are intercepted by the transaction interceptor
- Invocations within a bean do not lead to a transaction even if the invoked method is annotated with `@Transactional`

```
@Service
public BusinessService {
    public void businessMethod() {
        ...
        otherMethod();    // non-transactional invocation
    }
    @Transactional
    public void otherMethod() {
        ...
    }
}
```

# Transactional Settings

- The `@Transactional` annotation has the following properties

Name	Description
value	qualifier of the transaction manager to be used
propagation	propagation setting
isolation	isolation level
timeout	transaction timeout (in seconds)
readOnly	flag for read-only transaction
rollbackFor	array of exception classes that must cause rollback
noRollbackFor	array of exception classes that must not cause rollback

# Transactional Defaults

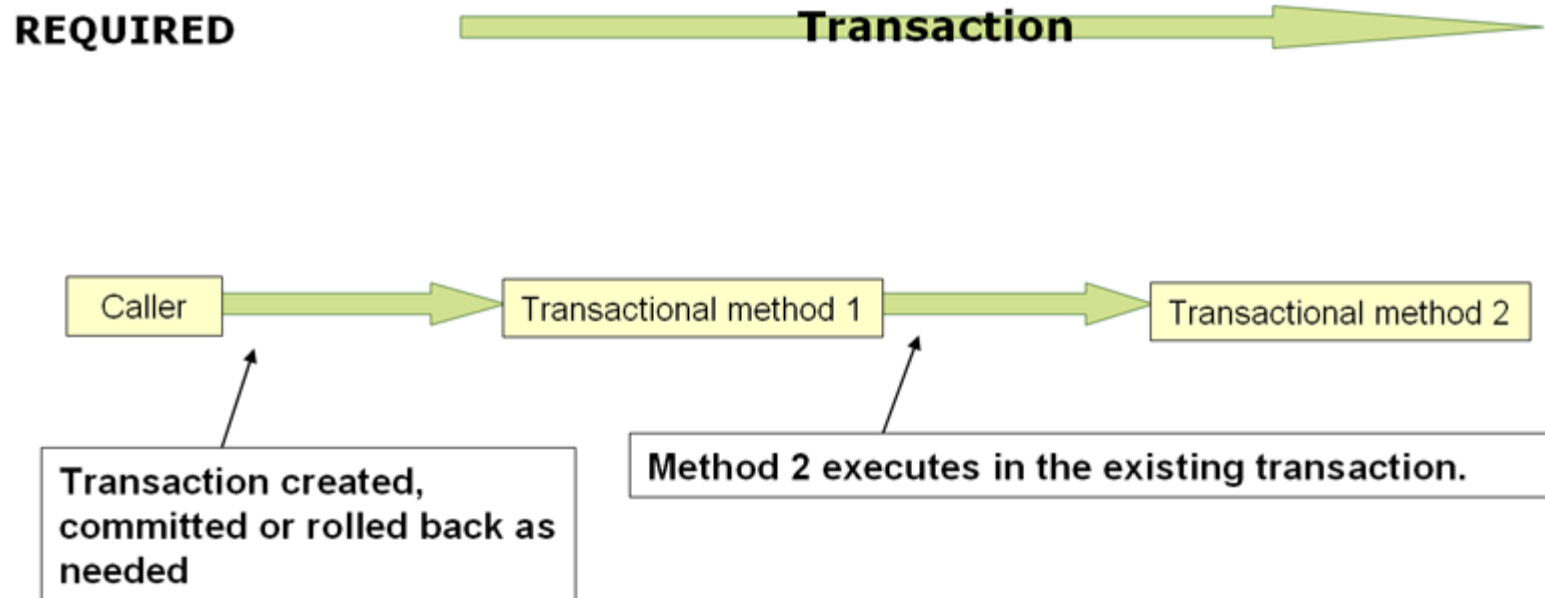
- With the default settings of the `@Transactional` annotation
  - the propagation setting is REQUIRED
  - the isolation level is DEFAULT
  - the transaction is read-write
  - the timeout is the default timeout of the underlying transaction system
  - runtime exceptions trigger rollback and checked exceptions do not

# Transaction Propagation

- When using transactions within nested method invocations, the propagation behavior of the transactions can be specified
- Nested methods may be executed in the same transaction scope or across multiple transactions
- Depending on the propagation, the outcome of an inner method may affect the transaction of outer methods

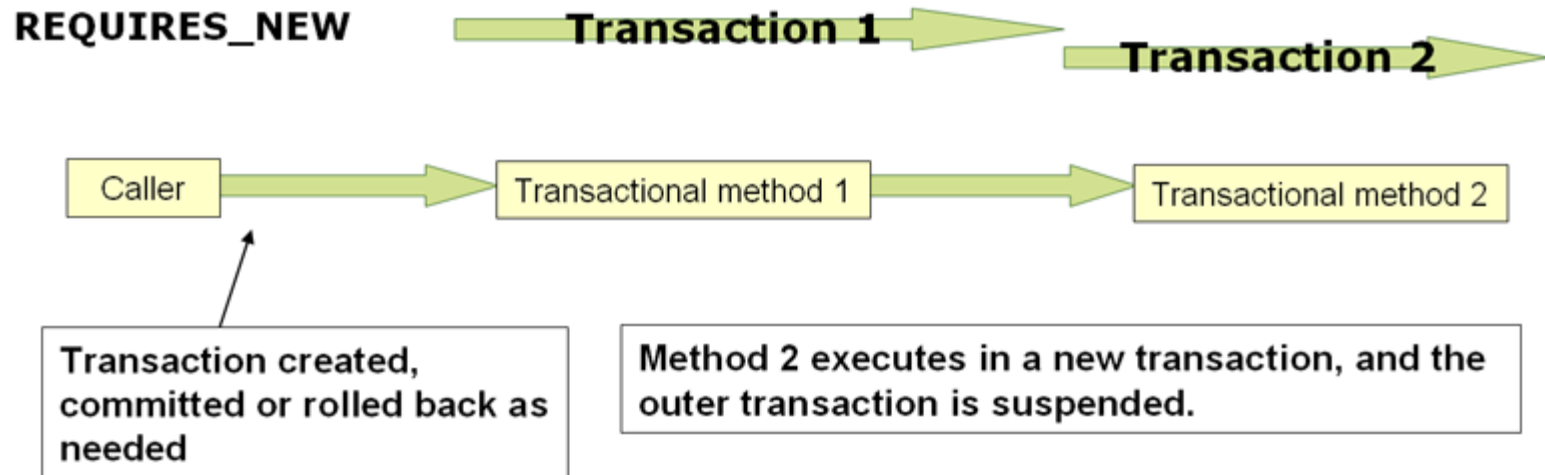
# The Required Propagation

- The REQUIRED propagation enforces a transaction, either by creating a new transaction or by participating in an existing outer transaction
- If an inner method causes a transaction to rollback, the outer method will fail to commit the transaction



# The RequiresNew Propagation

- The `REQUIRES_NEW` propagation always uses an independent transaction for the annotated method
- The inner transaction's rollback status does not affect an outer transaction
- Inner transactions can declare their own isolation level, timeout, and read-only settings



# Additional Propagations

Propagation	Description
SUPPORTS	Supports a current transaction, executes non-transactionally if none exists
NOT_SUPPORTED	Executes non-transactionally, suspends the current transaction if one exists
MANDATORY	Supports a current transaction, throws an exception if none exists
NEVER	Executes non-transactionally, throws an exception if a transaction exists
NESTED	Executes within a nested transaction if a current transaction exists

# Programmatic Transaction Management

- Spring provides the `TransactionTemplate` for programmatic transaction management
- The transactional application code can be passed as callback to the `execute` method
- The transaction can be rolled back by calling the `setRollbackOnly` method on the supplied `TransactionStatus` object

```
public class BusinessService {  
    @Autowired  
    private final TransactionTemplate txTemplate;  
  
    public void businessMethod() {  
        return txTemplate.execute(status -> {  
            try {  
                ...  
            } catch (BusinessException ex) { status.setRollbackOnly(); }  
        });  
    }  
}
```

# Programmatic Transaction Settings

- The transaction settings can be specified on the TransactionTemplate either programmatically or by configuration

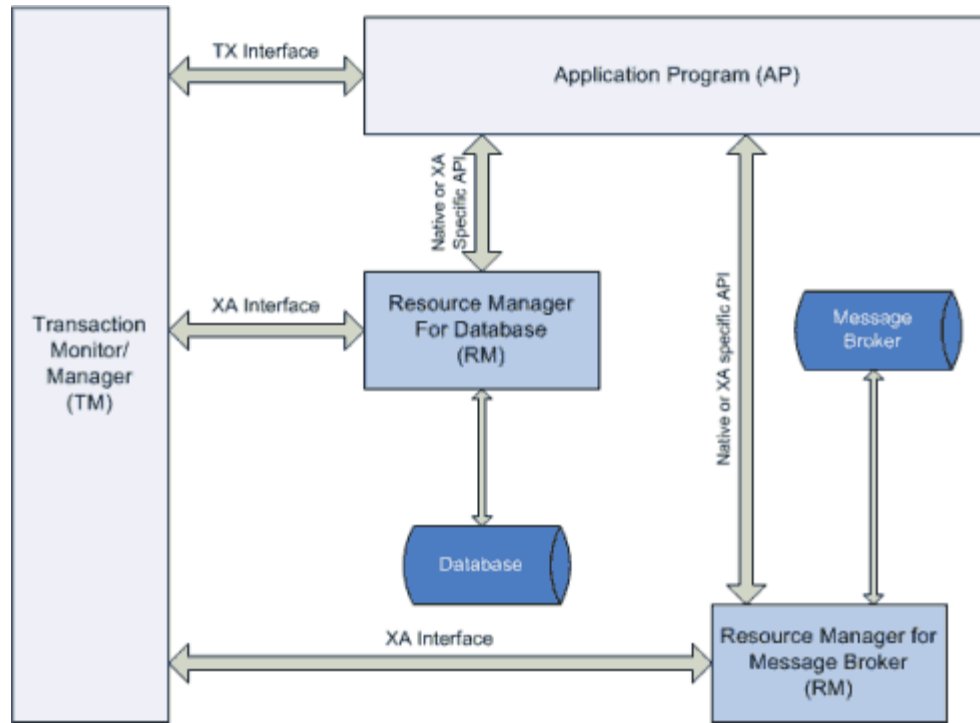
```
public class BusinessService {  
  
    private final TransactionTemplate txTemplate;  
  
    public BusinessService(PlatformTransactionManager transactionManager) {  
        txTemplate = new TransactionTemplate(transactionManager);  
        txTemplate.setIsolationLevel(ISOLATION_READ_UNCOMMITTED);  
        txTemplate.setTimeout(30);  
        ...  
    }  
}
```

# Distributed Transactions

# Introduction

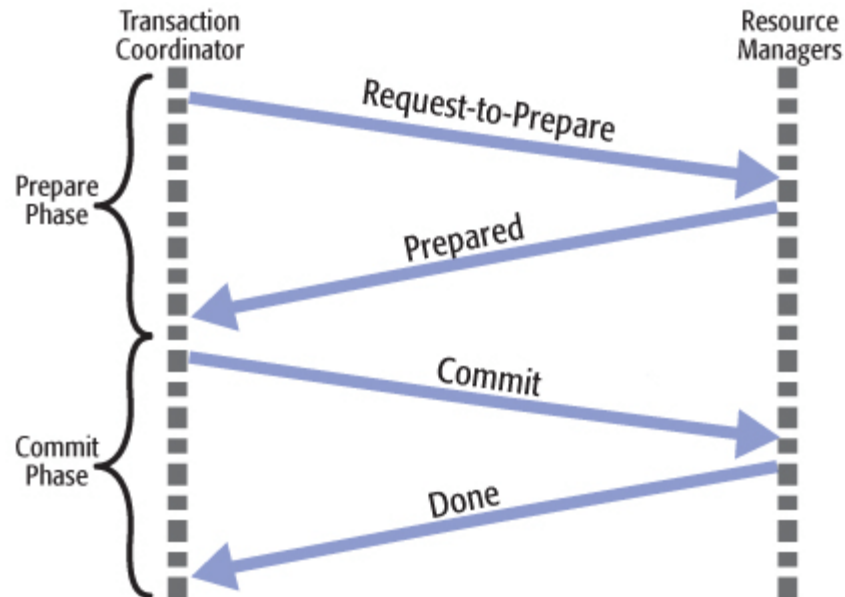
- A distributed transaction is a transaction that involves more than one transactional resource
- The *X/Open Distributed Transaction Processing (DTP)* model defines a standard architecture that allows applications to share resources and to coordinate their work in global transactions
- The *X/Open XA* interface enables resource managers to join transactions and to participate in the two phase commit protocol (2PC)

# Transaction Monitor



# Two-Phase Commit

- Distributed transactions are handled by a transaction manager
- During the preparation phase, the transaction manager instructs all resource managers to get ready to commit
- The resource managers inform the transaction manager if they approve the transaction
- In the commit phase, the transaction manager instructs all resource managers either to complete the commit or to rollback



# Java Transaction API

- The *Java Transaction API (JTA)* is a high-level API which consists of
  - an application interface for an application to define transaction boundaries
  - a transaction manager interface that allows an application to manage transactions
  - a Java mapping of the standard X/Open XA protocol

# Spring Distributed Transactions

- Spring Boot supports distributed JTA transactions across multiple XA resources by using an [Atomikos](#) embedded transaction manager
- When a JTA environment is detected, Spring uses the `JtaTransactionManager` to manage transactions and upgrades JMS, `DataSource` and JPA beans to support XA transactions
- The `@Transactional` annotation can be used to participate in a distributed transaction

