Spring Boot Transaction Management - Comprehensive Notes

1. Introduction to Transactions in Spring Boot

What are Transactions?

- **Definition**: A transaction is a sequence of database operations that are treated as a single unit of work
- **Purpose**: Ensures data integrity and consistency in database operations
- **Spring Boot Integration**: Provides seamless integration with Hibernate/JPA through declarative transaction management

Why Use Spring Transactions?

- **Simplified Management**: No need for manual transaction handling with Hibernate
- **Declarative Approach**: Use annotations instead of programmatic transaction code
- **Automatic Rollback**: Handles exceptions and rollbacks automatically
- **Connection Management**: Manages database connections efficiently

---

2. @Transactional Annotation

Basic Usage

- **Primary Annotation**: `@Transactional` is the main annotation for transaction management
- **Declarative Style**: Eliminates boilerplate transaction management code
- **Automatic Proxy Creation**: Spring creates proxies to handle transaction lifecycle

How @Transactional Works Internally

java

```java
// Spring wraps your method with transaction management code
createTransactionIfNecessary();
try {
    yourMethod(); // Your actual business logic
    commitTransactionAfterReturning();
} catch (Exception exception) {
    rollbackTransactionAfterThrowing();
    throw exception;
}
```

Application Levels (Priority Order - Lowest to Highest)

1. **Interface Level**: Applied to entire interface
2. **Superclass Level**: Applied to parent class
3. **Class Level**: Applied to entire class
4. **Interface Method Level**: Applied to specific interface method
5. **Superclass Method Level**: Applied to parent class method
6. **Method Level**: Applied to specific method (Highest Priority)

Example Implementation

```java
@Transactional(readOnly = true) // Class level - default read-only
public class EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepo;

    @Transactional // Method level - overrides class level settings
    public Employee addEmployee(Employee employee) {
        Employee savedEmployee = employeeRepo.save(employee);
        // Additional operations within same transaction
        return savedEmployee;
    }
}
```

---

3. ACID Properties of Transactions

A - Atomicity

- **All or Nothing Principle**: Either all operations succeed or all fail
- **Complete Execution**: All operations within transaction are executed successfully
- **Rollback Capability**: If any operation fails, entire transaction is rolled back
- **Data Integrity**: Prevents partial updates that could corrupt data

C - Consistency

- **State Transition**: Moves database from one consistent state to another
- **Constraint Validation**: All database constraints must be satisfied
- **Business Rules**: Ensures business logic rules are maintained
- **Data Validity**: Modified data passes all validation checks

I - Isolation

- **Transaction Separation**: Changes are invisible to other transactions until commit
- **Concurrent Access**: Multiple transactions can run simultaneously without interference
- **Data Protection**: Prevents dirty reads and inconsistent data access
- **Isolation Levels**: Different levels control visibility of uncommitted changes

D - Durability

- **Persistent Storage**: Committed changes are permanently saved
- **System Failure Recovery**: Data survives system crashes and restarts
- **Storage Guarantee**: Changes are written to non-volatile storage
- **Long-term Reliability**: Data remains intact over time

---

4. Transaction Propagation Types

REQUIRED (Default)

- **Behavior**: Join existing transaction or create new one
- **Usage**: Most common propagation type
- **Characteristics**:
  - Checks for active transaction
  - Creates new transaction if none exists
  - Joins existing transaction if available
  - Exception rolls back entire transaction (parent + child)

```java
@Transactional(propagation = Propagation.REQUIRED)
public void methodWithRequired() {
    // Business logic here
}
```

REQUIRES_NEW

- **Behavior**: Always create a completely new transaction
- **Usage**: When you need independent transaction handling
- **Characteristics**:
  - Suspends current transaction if exists
  - Creates new physical transaction with new database connection

- Outer transaction resumes after inner completes
- Independent rollback behavior

java

```java
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void methodWithRequiresNew() {
    // Runs in completely separate transaction
}
```

NESTED

- **Behavior**: Uses savepoints for nested operations
- **Usage**: When you need partial rollback capability
- **Characteristics**:
  - Creates savepoint if transaction exists
  - Rollback only to savepoint on exception
  - Works like REQUIRED if no active transaction
  - Allows partial transaction recovery

java

```java
@Transactional(propagation = Propagation.NESTED)
public void methodWithNested() {
    // Can rollback to savepoint if needed
}
```

MANDATORY

- **Behavior**: Requires existing transaction, throws exception if none
- **Usage**: When method must run within existing transaction context
- **Characteristics**:
  - Never creates new transaction
  - Throws exception if no active transaction
  - Joins existing transaction
  - Rollback affects entire transaction chain

java

```java
@Transactional(propagation = Propagation.MANDATORY)
public void methodWithMandatory() {
    // Must be called within existing transaction
}
```

NEVER

- **Behavior**: Throws exception if transaction exists
- **Usage**: When method must run outside transaction context
- **Characteristics**:
  - Prohibits transaction execution
  - Throws IllegalTransactionStateException if transaction exists
  - Internal methods can still create transactions
  - Useful for read-only operations that don't need transactions

java

```java
@Transactional(propagation = Propagation.NEVER)
public void methodWithNever() {
    // Must run without any transaction
}
```

NOT_SUPPORTED

- **Behavior**: Suspends existing transaction and runs without transaction
- **Usage**: When you need to temporarily escape transaction context
- **Characteristics**:
  - Suspends current transaction if exists
  - Executes without transaction
  - Resumes original transaction after completion
  - Useful for operations that shouldn't be part of transaction

java

```java
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public void methodWithNotSupported() {
    // Runs outside transaction context
}
```

SUPPORTS

- **Behavior**: Flexible - works with or without transaction
- **Usage**: When transaction is optional
- **Characteristics**:
  - Joins existing transaction if available
  - Runs without transaction if none exists
  - Most flexible propagation type
  - Adapts to calling context

```java
@Transactional(propagation = Propagation.SUPPORTS)
public void methodWithSupports() {
    // Works in any context
}
```

---

5. Transaction Isolation Levels

Overview

- **Purpose**: Control degree of isolation between concurrent transactions
- **Trade-off**: Balance between consistency and performance
- **Configuration**: Set via `@Transactional(isolation = Isolation.LEVEL)`

Read Phenomena

Dirty Read

- **Definition**: Reading uncommitted data from another transaction
- **Problem**: Data might be rolled back, making read invalid
- **Example**: T1 modifies data, T2 reads it, T1 rolls back
- **Risk**: Working with data that never officially existed

Non-Repeatable Read

- **Definition**: Same query returns different results within same transaction
- **Problem**: Data changes between reads in same transaction
- **Example**: T1 reads data, T2 modifies and commits, T1 reads again with different result
- **Risk**: Inconsistent data within single transaction

Phantom Read

- **Definition**: Same query returns different number of rows

- **Problem**: New rows appear/disappear between reads
- **Example**: T1 counts records, T2 inserts new record, T1 counts again with different total
- **Risk**: Set-based operations become unreliable

Isolation Levels

READ_UNCOMMITTED

- **Allows**: All read phenomena (Dirty, Non-repeatable, Phantom)
- **Performance**: Highest
- **Consistency**: Lowest
- **Use Case**: When performance is critical and consistency is less important

READ_COMMITTED (Default in most databases)

- **Prevents**: Dirty reads
- **Allows**: Non-repeatable reads, Phantom reads
- **Performance**: High
- **Consistency**: Moderate
- **Use Case**: Most common business applications

REPEATABLE_READ

- **Prevents**: Dirty reads, Non-repeatable reads
- **Allows**: Phantom reads
- **Performance**: Moderate
- **Consistency**: High
- **Use Case**: When consistent reads are important

SERIALIZABLE

- **Prevents**: All read phenomena
- **Allows**: None
- **Performance**: Lowest
- **Consistency**: Highest
- **Use Case**: Critical financial or audit operations

---

6. @Transactional Rules and Best Practices

Method Requirements

- **Visibility**: Must be applied only to `public` methods
- **Invocation**: Method must be called from outside the bean (external invocation)
- **Proxy Limitation**: Internal method calls bypass proxy and ignore @Transactional
- **Self-Invocation**: Calling @Transactional method from same class won't work

Best Practices

1. **Service Layer**: Apply transactions at service layer, not repository layer
2. **Method Granularity**: Keep transactional methods focused and small
3. **Exception Handling**: Be aware of checked vs unchecked exception rollback behavior
4. **Read-Only Optimization**: Use `readOnly = true` for query-only operations
5. **Timeout Settings**: Set appropriate timeout values for long-running operations
6. **Isolation Selection**: Choose appropriate isolation level based on business needs

Common Configuration Options

```java
@Transactional(
    propagation = Propagation.REQUIRED,
    isolation = Isolation.READ_COMMITTED,
    timeout = 30,
    readOnly = false,
    rollbackFor = {Exception.class},
    noRollbackFor = {BusinessException.class}
)
```

Performance Considerations

- **Connection Pooling**: Transactions hold database connections
- **Lock Duration**: Longer transactions increase lock contention
- **Resource Usage**: Consider memory and CPU impact
- **Batch Operations**: Use appropriate batch sizes for bulk operations

---

7. Advanced Transaction Concepts

Rollback Behavior

- **Default**: Rollback on unchecked exceptions (RuntimeException and Error)
- **Checked Exceptions**: Do not trigger rollback by default
- **Custom Configuration**: Use `rollbackFor` and `noRollbackFor` attributes

- **Manual Rollback**: Use `TransactionStatus.setRollbackOnly()`

Transaction Synchronization

- **Before Commit**: Execute code before transaction commits
- **After Commit**: Execute code after successful commit
- **After Rollback**: Execute code after rollback
- **Use Cases**: Cache invalidation, event publishing, cleanup operations

Testing Transactions

- **@Transactional in Tests**: Automatically rolls back after each test
- **@Rollback**: Control rollback behavior in tests
- **@Commit**: Force commit in test methods
- **Test Isolation**: Ensure tests don't interfere with each other

---

8. Common Pitfalls and Troubleshooting

Common Issues

1. **Self-Invocation**: Internal method calls bypass transaction proxy
2. **Private Methods**: @Transactional ignored on non-public methods
3. **Exception Handling**: Catching exceptions prevents rollback
4. **Connection Leaks**: Long-running transactions can exhaust connection pool
5. **Deadlocks**: Poor transaction design can cause database deadlocks

Debugging Tips

- **Enable Transaction Logging**: Set appropriate log levels
- **Monitor Connection Pool**: Watch for connection exhaustion
- **Profile Performance**: Identify slow transactions
- **Test Rollback Scenarios**: Verify exception handling works correctly

Monitoring and Metrics

- **Transaction Duration**: Monitor transaction execution time
- **Rollback Frequency**: Track rollback rates
- **Connection Usage**: Monitor database connection utilization
- **Lock Contention**: Identify database locking issues