**Configuring Transactions in Spring Boot**

Spring Boot automatically enables transaction management when using Spring Data JPA. However, you can explicitly enable it.

@**Configuration**

@**EnableTransactionManagement** // Enables declarative transaction management

public class TransactionConfig {

}

**Using @Transactional in the Service Layer**

**The @Transactional annotation ensures atomicity of database operations.**

## **Transaction Isolation Levels**

**READ\_UNCOMMITTED (Least Safe)**

* This is the lowest level of isolation and allows many simultaneous accesses.
* It allows **dirty reads, non-repeatable reads**, and **phantom reads**.
* It reads uncommitted data.
* **Fastest but risky**—not recommended for financial transactions
* **Use Case**: High performance, but minor inconsistencies allowed

**READ\_COMMITTED (DEFAULT Most Common)**

* Sees only committed data
* **Prevents dirty reads** but allows **non-repeatable reads** and **phantom reads**.
* Default in **most RDBMS (e.g., PostgreSQL, Oracle)**.
* **Use Case**: Most common for general applications (avoids dirty reads)

**REPEATABLE\_READ (Better Consistency)**

* It ensures consistency reads in one transaction
* It avoids **DIRTY READ** and **NON-REPEATABLE\_READ** and allows **Phantom Reads**
* **Use Case:** Banking, e-commerce transactions (ensures stable reads)

**SERIALIZABLE (Most Secure, Slowest)**

* It’s the highest level of isolation.
* It prevents all concurrency problems.
* Executes concurrent calls sequentially.
* It is expensive in terms of performance.
* Fully isolated transactions
* It avoids **DIRTY READ, PHANTOM READ** and **NON-REPEATABLE\_READ**
* Highest consistency but slowest due to locking
* **Use Case**: Critical financial transactions (full isolation)

**Dirty Reads** 🧐 → A transaction reads uncommitted data from another transaction.

**Non-Repeatable Reads** 🔄 → A transaction reads the same row twice, but the values change in between.

**Phantom Reads** 👻 → A transaction executes the same query twice, but the second execution returns new rows.

**@Transactional(isolation = Isolation.REPEATABLE\_READ)**

public void performTransactionalOperation() { // Business logic }

**Propagation & Isolation Levels in Transactions**

|  |  |
| --- | --- |
| **REQUIRED** (default) | Uses an existing transaction or starts a new one. |

|  |  |
| --- | --- |
| **REQUIRES\_NEW** | Always starts a new transaction, suspending any existing one. |

|  |  |
| --- | --- |
| **MANDATORY** | Requires an existing transaction, throws an error if none. |

|  |  |
| --- | --- |
| **SUPPORTS** | Uses a transaction if available, but does not create one. |

|  |  |
| --- | --- |
| **NOT\_SUPPORTED** | Executes outside of a transaction. |

|  |  |
| --- | --- |
| **NEVER** | Throws an error if called within a transaction. |

|  |  |
| --- | --- |
| **NESTED** | Runs a nested transaction within an existing transaction. |

**@Transactional(propagation = Propagation.REQUIRES\_NEW)**

public void updateEmployee(Employee employee) { employeeRepository.save(employee); }

**Handling Rollbacks & Exception Management**

By default, **Spring rolls back transactions for unchecked exceptions (RuntimeException & Error)** but commits for checked exceptions.

**🔹 Forcing Rollback on Specific Exception**

**@Transactional(rollbackFor = Exception.class)** // Rolls back on any Exception

public void processTransaction() throws Exception {

employeeRepository.save(new Employee("John"));

throw new Exception("Simulated failure! Rollback triggered.");

}

**🔹 Prevent Rollback for Certain Exceptions**

**@Transactional(noRollbackFor = IllegalArgumentException.class)**

public void processTransaction() {

employeeRepository.save(new Employee("John"));

throw new IllegalArgumentException("This won't trigger rollback.");

}

**Programmatic Transaction Management (Alternative to @Transactional)**

private final TransactionTemplate transactionTemplate;

public void addEmployee(Employee emp1, Employee emp2) {

transactionTemplate.execute(new TransactionCallbackWithoutResult() {

@Override protected void doInTransactionWithoutResult(TransactionStatus status) { try { employeeRepository.save(emp1); employeeRepository.save(emp2);

} catch (Exception e) {

status.setRollbackOnly(); // Manually trigger rollback } } }); } }

Transactions in MongoDB (NoSQL)

MongoDB does **not support ACID transactions** by default but allows transactions in **replica sets**.

**@Transactional**

public void performTransaction() {

try (ClientSession session = mongoTemplate.getMongoDbFactory().getSession()) {

**session.startTransaction();**

mongoTemplate.save(new Employee("Alice"));

mongoTemplate.save(new Employee("Bob"));

**session.commitTransaction();** // Ensure commit } catch (Exception e) {

throw new RuntimeException("Transaction failed! Rolling back."); } }

**Two-Phase Commit (2PC)**

* **Two-Phase Commit** (2PC) is a distributed transaction protocol used to ensure data consistency across multiple microservices or databases.
* It is typically used when multiple services need to participate in a single transaction and must either **commit** or **roll back** together.

**Phase 1: Prepare Phase**

1. The **coordinator service** (Transaction Manager) sends a **prepare request** to all participating microservices.
2. Each microservice executes the transaction **without committing it**.
3. Each microservice responds with either:

**"YES" (Ready to Commit)** if it successfully executes its part.

**"NO" (Abort)** if it encounters any issues.

**Phase 2: Commit/Rollback Phase**

1. If **all participants** respond with "YES":
   * The coordinator sends a **commit request** to all participants.
   * Each participant **commits** the transaction.
2. If **any participant** responds with "NO":
   * The coordinator sends a **rollback request** to all participants.
   * Each participant **rolls back** its transaction to maintain consistency.

**Challenges with 2PC in Microservices**

1. **Performance Overhead** - 2PC is slow due to multiple network calls.
2. **Single Point of Failure** - If the transaction coordinator fails, transactions can be left in an uncertain state.
3. **Scalability Issues** - In microservices, where services are loosely coupled, 2PC can introduce dependencies between services.

@Service

public class TransactionService {

@Autowired

private RepositoryOne repositoryOne;

@Autowired

private RepositoryTwo repositoryTwo;

@Transactional

public void performTransaction() {

repositoryOne.save(new EntityOne("Data 1"));

repositoryTwo.save(new EntityTwo("Data 2"));

// If any exception occurs, both transactions will be rolled back

}

}

**1. Optimistic Locking (Using @Version)**

Optimistic locking is used when **multiple transactions** read the same data and try to update it, ensuring **data integrity** using version numbers.

**Steps:**

1. Add @Version field in the entity.
2. Handle OptimisticLockException in case of concurrent updates.

@Entity

@Getter

@Setter

public class Account {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String accountNumber;

private double balance;

@Version // Enables Optimistic Locking

private int version;

}

**2. Pessimistic Locking (Using @Lock)**

Pessimistic Locking prevents **multiple transactions** from modifying data simultaneously by locking the record in the database.

**Steps:**

1. Use @Lock(LockModeType.PESSIMISTIC\_WRITE) to prevent concurrent updates.
2. Handle locking exceptions.

import org.springframework.data.jpa.repository.Lock;

@Repository

public interface AccountRepository extends JpaRepository<Account, Long> {

**@Lock(LockModeType.PESSIMISTIC\_WRITE)** // Lock the record

@Query("SELECT a FROM Account a WHERE a.id = :id")

Account findByIdWithLock(@Param("id") Long id);

}

| **Feature** | **Optimistic Locking** | **Pessimistic Locking** |
| --- | --- | --- |

|  |  |  |
| --- | --- | --- |
| Lock Type | Version-based | Database Lock |

|  |  |  |
| --- | --- | --- |
| Concurrency | High | Low |

|  |  |  |
| --- | --- | --- |
| Performance Impact | Lower | Higher |

|  |  |  |
| --- | --- | --- |
| When to Use | Frequent reads, rare updates | High contention updates |

|  |  |  |
| --- | --- | --- |
| Exception Handling | OptimisticLockException | PessimisticLockException |
|  |  |  |

**Testing**

1. **Optimistic Locking Test:**
   * Open two terminals.
   * Fetch the same account and update it simultaneously.
   * One transaction will fail with OptimisticLockException.
2. **Pessimistic Locking Test:**
   * Open two terminals.
   * Try updating the same account simultaneously.
   * The second request will wait until the first transaction is completed.