

## ✓ What is `@Transactional` in Spring?

`@Transactional` is an annotation used to manage **transactions** in Spring. It allows you to wrap a method in a **database transaction**—which means all operations inside it will succeed or fail **as one unit**.

## 🔥 Why Use Transactions?

- Ensure **data integrity**.
- Avoid **partial updates** if something fails.
- Automatically **rollback** if an exception occurs.

## Concepts You Must Know

Concept	Meaning
Transaction	A group of DB operations treated as a single unit
Commit	Save changes to the DB permanently
Rollback	Undo changes if an error happens
ACID	Atomicity, Consistency, Isolation, Durability

- **Atomicity:** All steps in a transaction happen completely or not at all.
- **Consistency:** The database stays valid before and after the transaction.
- **Isolation:** Transactions don't affect each other while running at the same time.
- **Durability:** Once a transaction is committed, its data stays even after a crash.

## ✨ **A = Atomicity**

All or nothing:

If a process has 3 steps, and step 2 fails, the whole thing is canceled.

✓ All steps done — OR ✗ nothing is saved.

## 🔒 **I = Isolation**

No interference:

If 2 people update the same data at once, their actions don't mess each other up.

Each one runs like it's alone.

## ✓ **C = Consistency**

Always valid data:

Data must follow rules. If something breaks a rule, it won't be saved.

Example: Can't save an account with -\$100 if rule says  $\text{balance} \geq 0$ .

## 🧱 **D = Durability**

Saved forever:

Once you save, it's safe — even if power goes off or system crashes.

# start write code

## ✓ What `CommandLineRunner` Does:

When you implement `CommandLineRunner`, the method `run(String... args)` will be executed **automatically** once the application context is loaded, **before the application fully finishes startup**.

## ? What if `@Transactional` is NOT Present?

Without `@Transactional`, each database operation (like `repo.save()`) is committed **IMMEDIATELY** after it's called.

So, if you save an entity and then throw an exception—that save is **already committed** and will **NOT** be rolled back.

difference between adding and not adding **@Transactional**

```
public void transferWithRuntimeException(Long fromId, Long toId, double amount) {  
    BankAccount from = repo.findById(fromId).orElseThrow();  
    BankAccount to = repo.findById(toId).orElseThrow();  
  
    from.setBalance(from.getBalance() - amount);  
    to.setBalance(to.getBalance() + amount);  
  
    repo.save(from);  
    repo.save(to);  
  
    // Unchecked exception => causes rollback  
    throw new RuntimeException("Something went wrong after saving!");  
}
```

✓ If `@Transactional` is present:

java

Copy Edit

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

- Spring creates a transaction at the beginning of the method.
- All database operations ( `findById` , `save` ) run inside the same transaction.
- When `RuntimeException` is thrown:
  - Spring rolls back the entire transaction.
  - No changes will be saved in the database (both balances stay unchanged).



## ✗ If `@Transactional` is not present:

java

Copy Edit

```
public void transferWithRuntimeException(...) { ... }
```

- Each `repo.save(...)` is executed **outside of any transaction** or using **default behavior**, which is usually **auto-commit per save**.
- When the exception is thrown:
  - The first two `save` calls **already committed** to the database.
  - But the exception still happens, **after saving**.
  - This leads to **inconsistent state**: money might be deducted from `from` account but not added to `to`.

### ✓ Case 1: `transferWithRuntimeException` WITHOUT `@Transactional`

java

Copy Edit

```
public void transferWithRuntimeException(Long fromId, Long toId, double amount) {  
    BankAccount from = repo.findById(fromId).orElseThrow();  
    BankAccount to = repo.findById(toId).orElseThrow();  
  
    from.setBalance(from.getBalance() - amount);  
    to.setBalance(to.getBalance() + amount);  
  
    repo.save(from); // saved immediately  
    repo.save(to);   // saved immediately  
  
    throw new RuntimeException("Runtime error after saving");  
}
```

#### ⚠ Result:

- Alice's balance is **updated and saved**.
- Bob's balance is **updated and saved**.
- Even though a `RuntimeException` is thrown → **NO rollback** happens.
- **Both changes are committed** to the database!

## ✗ Case 2: `transferWithCheckedException` WITHOUT `@Transactional`

java

Copy Edit

```
public void transferWithCheckedException(Long fromId, Long toId, double amount) throws Exception {  
    BankAccount from = repo.findById(fromId).orElseThrow();  
    BankAccount to = repo.findById(toId).orElseThrow();  
  
    from.setBalance(from.getBalance() - amount);  
    to.setBalance(to.getBalance() + amount);  
  
    repo.save(from); // saved immediately  
    repo.save(to);  // saved immediately  
  
    throw new Exception("Checked exception after saving");  
}
```

### ⚠ Result:

- Same as above — both saves happen immediately.
- Checked exception is thrown **after the save**.
- **No rollback**, because no transaction exists.
- The data is already persisted.

## Default Behavior of `@Transactional`

- By default, Spring rolls back on **unchecked exceptions** (subclasses of `RuntimeException`).
- For **checked exceptions**, it won't rollback unless you configure it explicitly.

### ✓ 1. Unchecked Exceptions (Default: Rolls back)

Unchecked = `RuntimeException` (or any subclass)  
Spring will **automatically rollback** the transaction.

### ✓ 2. Checked Exceptions (Default: DOES NOT rollback)

Checked = Exception that must be caught or declared  
Spring won't rollback by default, unless you tell it to using `rollbackFor`.

## ✓ Example 1 – Unchecked Exception (Rollback Happens Automatically)

```
java                                                                    Copy Edit

@Service
public class BankService {

    @Autowired
    private BankAccountRepository repo;

    @Transactional
    public void transferWithRuntimeException(Long fromId, Long toId, double amount) {
        BankAccount from = repo.findById(fromId).orElseThrow();
        BankAccount to = repo.findById(toId).orElseThrow();

        from.setBalance(from.getBalance() - amount);
        to.setBalance(to.getBalance() + amount);

        repo.save(from);
        repo.save(to);

        // Unchecked exception => causes rollback
        throw new RuntimeException("Something went wrong after saving!");
    }
}
```

📄 Result: NO money is transferred — rollback happens.

## ✗ Example 2 – Checked Exception (Rollback Does NOT Happen)

```
java                                                                    Copy Edit

@Service
public class BankService {

    @Autowired
    private BankAccountRepository repo;

    @Transactional
    public void transferWithCheckedException(Long fromId, Long toId, double amount) throws Exception {
        BankAccount from = repo.findById(fromId).orElseThrow();
        BankAccount to = repo.findById(toId).orElseThrow();

        from.setBalance(from.getBalance() - amount);
        to.setBalance(to.getBalance() + amount);

        repo.save(from);
        repo.save(to);


        // Checked exception => does NOT rollback by default
        throw new Exception("Checked exception thrown after saving!");
    }
}
```

🔍 Result: Money **WILL** be transferred even though an error happened!



## Advanced Options

java

 Copy  Edit

```
@Transactional(  
    rollbackFor = Exception.class,          // Rollback on ANY exception  
    noRollbackFor = ArithmeticException.class // Won't rollback if ArithmeticException  
)
```

## What is Propagation?

Propagation defines how transactions behave when one transactional method calls another.

For example:

- You're in `transfer()`, which is `@Transactional`
- It calls another method like `logTransfer()`, which is also `@Transactional`

💡 Should `logTransfer()`:

- Join the current transaction?
- Start a new one?
- Suspend the existing one?

That's what **Propagation** controls.



## ✓ All Propagation Types (with Real Examples)

Type	Description	Real-world Analogy
REQUIRED	Join existing or start new	"Join the current group, or start a new one if none exists"
REQUIRES_NEW	Always start new	"Leave your team and work solo"
NESTED	New savepoint inside current	"Sub-task inside a task, can be undone separately"
SUPPORTS	Join if exists, else no TX	"Help only if a group is working"
NOT_SUPPORTED	Suspend if exists	"Pause the group while I work non-transactionally"
NEVER	Throw error if TX exists	"I refuse to work if a group exists"
MANDATORY	Must join existing TX	"Work only if a group already exists"

## ✓ 1. REQUIRED (Default)

java

Copy Edit

```
@Service
public class AuditService {
    @Transactional(propagation = Propagation.REQUIRED)
    public void logTransfer(String message) {
        System.out.println("Logging: " + message);
        // Will use the same transaction from caller
    }
}
```

java

Copy Edit

```
@Service
public class BankService {
    @Autowired
    private AuditService audit;

    @Transactional
    public void transfer(Long fromId, Long toId, double amount) {
        // withdraw, deposit ...
        audit.logTransfer("Transfer completed");

        throw new RuntimeException("Oops!"); // rollback both transfer and log
    }
}
```

- **Result: Both rolled back because they share the same transaction.**

## ✓ 2. REQUIRES\_NEW

java

Copy Edit

```
@Service
public class AuditService {
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    public void logTransfer(String message) {
        System.out.println("Logging: " + message);
    }
}
```

java

Copy Edit

```
@Service
public class BankService {
    @Autowired
    private AuditService audit;

    @Transactional
    public void transfer(Long fromId, Long toId, double amount) {
        // update balances...

        audit.logTransfer("Transfer done");

        throw new RuntimeException("Error in main transaction");
    }
}
```

● Result:

- Audit is committed (separate TX)
- Main transfer rolls back

### ✓ 3. NESTED

java

Copy Edit

```
@Service
public class AuditService {
    @Transactional(propagation = Propagation.NESTED)
    public void logTransfer(String message) {
        throw new RuntimeException("Log failed"); // only nested tx rollback
    }
}
```

java

Copy Edit

```
@Service
public class BankService {
    @Autowired
    private AuditService audit;

    @Transactional
    public void transfer(Long fromId, Long toId, double amount) {
        // balances update...

        try {
            audit.logTransfer("Transfer recorded");
        } catch (Exception e) {
            System.out.println("Log failed, continue transfer");
        }

        // continues and commits
    }
}
```

## NESTED: "Savepoint inside current transaction"

### ■ Meaning:

- It creates a **savepoint** inside the existing transaction.
- If an error happens in the nested method, it can **rollback itself** without affecting the outer transaction.

### 💡 Think of it like:

You're writing a Word document (main TX), and you add a section (nested). If you undo the section, the rest of the document stays.

#### ✓ 4. SUPPORTS

java

Copy Edit

```
@Service
public class AuditService {
    @Transactional(propagation = Propagation.SUPPORTS)
    public void logTransfer(String message) {
        System.out.println("Log: " + message);
    }
}
```

java

Copy Edit

```
@Service
public class BankService {
    public void transferWithoutTx() {
        // no @Transactional
        audit.logTransfer("No transaction context");
    }

    @Transactional
    public void transferWithTx() {
        audit.logTransfer("Runs inside existing transaction");
    }
}
```



### ■ Meaning:

- If called inside a transaction, it joins it.
- If called without a transaction, it runs normally (non-transactionally).

💡 Think of it like:

"I'll help if the team (transaction) is already working, otherwise I work alone."

### ● Result:

- Joins existing transaction if called from one
- Runs without transaction otherwise





























