@Transactional

Internal Working Cheat Sheet

Most bugs are not SQL bugs — they are transaction boundary bugs.

@Transactional wraps an eligible method call in a transaction boundary using Spring AOP. It manages begin, suspend, resume, commit, rollback, flush timing, and exception-based rollback rules — but it does **NOT** rewind your Java objects automatically.

What it really controls

1. DB transaction lifecycle
2. Persistence context scope
3. Flush / commit / rollback timing
4. Propagation across proxied method calls

What it does NOT control

1. Local variable history
2. Object reference reassignment
3. In-memory state reset after rollback
4. Self-invoked method interception

In this carousel

Proxy rules | Entity state | Args & returns | Rollback | Propagation | Production patterns

SH Sajid H
linkedin.com/in/sajid-h-2617a110b

How @Transactional actually starts

Preconditions

- Method call must pass through Spring proxy
- Usually **public** method on Spring bean
- TransactionManager must be configured
- Annotation can be on class or method
- Self-invocation bypasses advice

1. External caller invokes bean
2. Proxy resolves transaction attributes
3. TransactionInterceptor opens / joins tx
4. Connection + persistence context bound to thread
5. Target method executes
6. Commit or rollback, then cleanup

Typical flow

```
@Service
class OrderService {
   @Transactional
   public void placeOrder(OrderRequest req) {
      orderRepo.save(...);
      inventoryRepo.reserve(...);
   }
}
```

If the call comes from another bean through the proxy, Spring wraps this method in a transaction.

No transaction advice when...

- private / final / static method
- constructor call
- same-class self call
- manually new OrderService()

Rule: annotation alone does nothing — interception is the key.

SH Sajid H

linkedin.com/in/sajid-h-2617a110b

Persistence context & dirty checking

Managed objects • Entity state • Flush behavior

New
not tracked

→

Managed
tracked by persistence context

→

Detached
plain object now

→

Removed
scheduled for delete

What dirty checking means

- Inside an active transaction, managed entities are watched for field changes
- Changed fields are synchronized during flush / commit
- You often do NOT need repo.save() after modifying a managed entity
- Detached objects are NOT auto-persisted

```
@Transactional
public void activateUser(Long id) {
   User u = repo.findById(id).orElseThrow();
   u.setStatus(Status.ACTIVE);   // dirty checking
}
```

Because 'u' is managed, Hibernate updates it at flush/commit time.

Managed entity changed inside tx	Detached entity changed outside tx
↓	↓
UPDATE on flush/commit	No DB update unless merged/saved

Flush is not the same as commit

- flush = send SQL
- commit = make transaction permanent
- rollback can still undo flushed SQL before commit

Remember: @Transactional governs DB state, not automatic Java object rollback.

SH Sajid H | linkedin.com/in/sajid-h-2617a110b

Passing objects, assignments, and mutation

Java references vs transaction behavior

Java passes object references by value.
Spring transactions do not clone your objects.

A Mutate passed object

```
void f(UserDto dto) {
   dto.setName('A');
}
```

Outcome

- Caller sees changed fields
- Rollback does NOT revert dto in memory
- Only DB work is rolled back

B Reassign parameter

```
void f(UserDto dto) {
   dto = new UserDto();
   dto.setName('B');
}
```

Outcome

- Caller reference unchanged
- Local reassignment affects only local variable

C Pass detached entity into @Transactional

```
@Transactional
void update(User u) {
   u.setStatus(ACTIVE);
}
```

Outcome

- If 'u' is detached, change may not be persisted
- Load managed entity or merge/save explicitly

What changes where?

Local variable assignment	→	Method scope only
Field mutation on shared object	→	Visible to caller
Managed entity mutation in tx	→	Persisted by dirty checking
Detached entity mutation	→	In-memory only unless merged

Production-safe pattern

```
@Transactional
public void updateName(Long id, String name) {
   User managed = repo.findById(id).orElseThrow();
   managed.setName(name);
}
```

Prefer IDs + values over passing detached entities across layers.

Rollback reverts database effects,
not arbitrary object mutations in memory.

SH Sajid H | linkedin.com/in/sajid-h-2617a110b

Returns, detached state, and lazy loading

What happens after the transaction ends?

When a transactional method returns, its managed entities usually leave the persistence context and become detached.

A Return entity

- May work for already-loaded fields
- Lazy associations can fail later
- Caller now holds a detached object

B Return DTO

- Safe for API/service boundaries
- No lazy-loading surprises
- Best for clear contracts

C Return primitive / boolean / count

- No persistence context concern
- Simple and stable

Bad

```
@Transactional
public Order getOrder(Long id) {
    return repo.findById(id).orElseThrow();
}
```

Better

```
@Transactional(readOnly = true)
public OrderDto getOrder(Long id) {
    Order o = repo.findById(id).orElseThrow();
    return mapper.toDto(o);
}
```

LazyInitializationException risk

- Returned entity accessed after tx
- Lazy association not initialized
- No open session / persistence context
- Fetch what you need or map to DTO

OSIV note

Open Session in View can hide lazy-loading issues, but many teams disable it in production for cleaner boundaries.

Return DTOs across boundaries; keep entities inside transactional business logic.

SH Sajid H | linkedin.com/in/sajid-h-2617a110b

Rollback rules & exception behavior

Runtime vs checked exceptions • rollback-only traps

Default Spring behavior

- Rollback on RuntimeException
- Rollback on Error
- Checked exceptions do **NOT** rollback by default

Scenario	Result
throw new RuntimeException()	rollback
throw new IOException()	commit unless rollbackFor used
@Transactional(rollbackFor = Exception.class)	checked exceptions rollback too
@Transactional(noRollbackFor = CustomException.class)	commits for that exception

A

```
@Transactional(rollbackFor = Exception.class)
public void importFile() throws Exception {
   repo.save(...);
   throw new IOException();
}
```

Checked exception now triggers rollback.

B Swallowed exception trap

```
@Transactional
public void process() {
   try {
      repo.save(...);
      risky();
   } catch (Exception e) {
      log.error('handled', e);
   }
}
```

If exception is fully swallowed, tx may commit

Re-throw or mark rollback-only when needed

UnexpectedRollbackException

If an inner operation marks the transaction rollback-only, the outer method may continue and then fail at commit with **UnexpectedRollbackException**.

Don't hide exceptions accidentally

Use rollbackFor intentionally

Keep transactional error policy explicit

SH Sajid H

linkedin.com/in/sajid-h-2617a110b

Propagation modes that matter in production

REQUIRED • REQUIRES_NEW • NESTED and friends

REQUIRED
join existing or create new

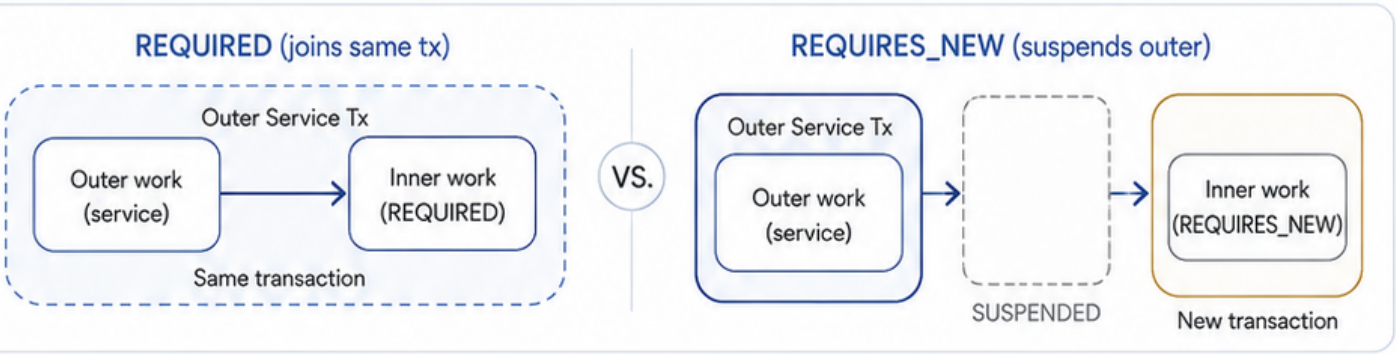
REQUIRES_NEW
suspend outer, start new

NESTED
savepoint within existing tx (if supported)

SUPPORTS
run with tx if present

NOT_SUPPORTED
run without tx

MANDATORY
must already have tx



Production scenarios

Audit log that must persist even if outer fails	→ REQUIRES_NEW
Normal service-to-service repository work	→ REQUIRED
Partial rollback with savepoint style logic	→ NESTED (DB/manager support needed)

Use REQUIRES_NEW carefully

- Creates independent commit
- Can increase connection usage
- May surprise outer-flow expectations

```
@Transactional
public void placeOrder() {
    orderService.save();
    auditService.logInNewTx();
}

@Transactional(propagation = Propagation.REQUIRES_NEW)
public void logInNewTx() { ... }
```

Choose propagation based on business boundary, not habit.

SH Sajid H | linkedin.com/in/sajid-h-2617a110b

Isolation, readOnly, and flush timing

Concurrency behavior • optimization hints • SQL emission

Isolation levels

Choose the weakest level that safely fits

READ_COMMITTED	good default; avoids dirty reads
REPEATABLE_READ	stable re-reads in a transaction
SERIALIZABLE	strongest isolation, lowest concurrency
DEFAULT	use database default

Isolation reduces anomalies - it does not replace locking.

readOnly = true

Intent and optimization hint

- Communicates read-only intent
- May enable ORM or driver optimizations
- Not a security boundary
- Do not rely on it to block writes everywhere

```
@Transactional(readOnly = true)
public List<OrderDto> findOpenOrders() { ... }
```

When SQL may be flushed

- Before commit
- Before certain queries
- On explicit flush()
- Still rollback-able until commit

flush = send SQL; commit = make it permanent

Flush + concurrency

```
entityManager.flush();
// sends pending SQL now
```

- Long transactions reduce throughput
- Use optimistic or pessimistic locking intentionally

Production rule

Keep the transaction short. Do not hold database locks while waiting for remote REST calls, file I/O, or slow user interactions.

Small transactions + correct isolation + explicit intent = safer production behavior.

SH Sajid H

in linkedin.com/in/sajid-h-2617a110b

Common production pitfalls

What breaks most often in real systems

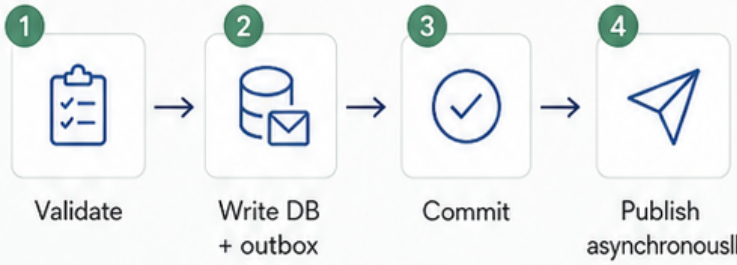
Pitfall	Safer pattern
Self-invocation of @Transactional method	Split into another bean or call via proxy
Long transaction around remote REST call	Keep tx short; commit DB work before remote I/O
Publishing message directly with DB write	Use outbox pattern for reliability
Returning entities to controller	Return DTOs
Catching and hiding exceptions	Re-throw or mark rollback-only intentionally

Looks fine, fails in prod

```
@Transactional
public void checkout() {
   repo.save(order);
   paymentClient.charge(...); //
   // remote call inside tx
}
```

⚠ Locks stay open while waiting on network.

Safer flow



Async / event boundary

- @Async runs on another thread
- Thread-bound tx context does not magically flow
- Be explicit about boundaries

Most @Transactional problems are boundary-design problems.

SH Sajid H | linkedin.com/in/sajid-h-2617a110b

@Transactional

Decision matrix & final checklist

What to remember before coding or debugging

Need	Use	Why
Normal service method	`REQUIRED`	default business transaction
Independent audit / retry record	`REQUIRES_NEW`	separate commit
Read-only query	`readOnly = true`	clear intent + possible optimization
API / controller response	`DTO`	avoid detached / lazy issues
Checked exception should rollback	`rollbackFor`	default rule is not enough

10 rules to remember

1. No proxy call = no transactional advice
2. Self-invocation bypasses interception
3. Managed entities are dirty-checked
4. Detached objects are plain objects
6. Rollback affects DB, not Java object history
7. Flush is not commit
7. Runtime exceptions rollback by default
8. Keep transactions short
9. Prefer DTOs at boundaries
10. Choose propagation intentionally

When something is weird, check...

- Was the method actually proxied?
- Did an exception get swallowed?
- Is the entity managed or detached?
- Did REQUIRES_NEW / rollback-only change behavior?
- Is lazy loading happening outside the tx?

Understand the boundary, and @Transactional becomes predictable.

SH Sajid H

linkedin.com/in/sajid-h-2617a110b