Transaction Isolation Levels with Spring Boot

Transaction isolation levels are one of those database topics many Spring Boot developers know by name, but not always by impact.

The impact can be huge.

A transaction is not only about commit and rollback. It is also about visibility.

Isolation answers this question: what can one transaction see while another transaction is still running?

In Spring Boot, isolation can be defined with `@Transactional`.

```
@Transactional(isolation = Isolation.READ_COMMITTED)
public void updateAccountBalance(Long accountId, BigDecimal amount) {
    Account account = accountRepository.findById(accountId).orElseThrow();
    account.debit(amount);
}
```

Many teams use the default isolation level without knowing what the database default actually is.

That is dangerous because different databases may behave differently.

READ_UNCOMMITTED

A transaction may read changes made by another transaction before they are committed. This can cause dirty reads.

In a fintech payments platform, this is rarely appropriate for money movement, ledger entries, account balances, settlement state, reconciliation, or risk decisions.

Example problem: Transaction A updates an account balance but does not commit yet. Transaction B reads that new balance. Transaction A rolls back. Transaction B just used data that never really existed.

Possible use case: non-critical operational dashboards where approximate data is acceptable, such as a temporary internal screen showing estimated queue depth or rough processing volume. Even there, most teams should prefer a replica, analytics store, cache, or event stream instead of dirty reads.

```
@Transactional(isolation = Isolation.READ_UNCOMMITTED, readOnly = true)
public PaymentOpsSnapshot getApproximateOpsSnapshot() {
    return paymentOpsRepository.loadApproximateSnapshot();
}
```

READ_COMMITTED

A transaction can only read committed data. This prevents dirty reads.

This is often a good default for standard payment platform operations where each operation validates the latest committed state and writes a small, well-defined change.

Fintech payments example: marking a payment as pending, inserting an audit event, storing a received webhook, or updating a payment attempt after receiving a response from a provider.

It is usually enough when the operation does not require the same row or query result to remain unchanged throughout a longer calculation.

```
@Transactional(isolation = Isolation.READ_COMMITTED)
public void recordProviderWebhook(ProviderWebhook webhook) {
    Payment payment = paymentRepository.findByProviderRef(webhook.providerRef())
```

```
            .orElseThrow();

    payment.applyProviderStatus(webhook.status());
    paymentEventRepository.save(PaymentEvent.from(webhook));
}
```

Risk: another committed transaction may change data between two reads inside the same transaction. This can cause non-repeatable reads.

REPEATABLE_READ

If a transaction reads the same row twice, it should see the same data both times.

In a fintech payments platform, this is useful when a decision depends on a stable view of rows read early in the transaction.

Fintech payments example: calculating fees, limits, or invoice totals based on a set of committed payment records that should not change during the calculation.

Another example: validating a merchant payout batch where the merchant configuration and payout instructions must stay stable during the operation.

```
@Transactional(isolation = Isolation.REPEATABLE_READ)
public PayoutBatchPreview calculatePayoutBatch(Long merchantId) {
    Merchant merchant = merchantRepository.findById(merchantId).orElseThrow();

    List<Payment> settledPayments = paymentRepository
            .findSettledPaymentsReadyForPayout(merchantId);

    return payoutCalculator.preview(merchant, settledPayments);
}
```

Cost: stronger isolation may increase contention. Keep the transaction short and avoid external calls inside it.

SERIALIZABLE

This is the strongest common isolation level. Transactions behave as if they executed one after another.

In a fintech payments platform, this should be reserved for operations where accepting a race condition is more expensive than reduced throughput.

Fintech payments example: reserving the last available balance from a prefunded account when multiple payment requests can arrive at the same time.

Another example: enforcing a strict daily transfer limit when concurrent payment attempts must not slip through the same remaining limit window.

It can protect critical business invariants, but it should not be used globally.

```
@Transactional(isolation = Isolation.SERIALIZABLE)
public void reserveAvailableBalance(Long accountId, BigDecimal amount) {
    Account account = accountRepository.findById(accountId).orElseThrow();

    if (account.getAvailableBalance().compareTo(amount) < 0) {
        throw new InsufficientFundsException();
    }

    account.reserve(amount);
}
```

Cost: more blocking, more retries, possible serialization failures, and lower throughput. Use it only for the small critical section that protects the business invariant.

Isolation level choice in a payments platform

- READ_UNCOMMITTED: almost never for financial correctness; only for approximate, non-critical operational views, and preferably avoid it.
- READ_COMMITTED: good default for many normal payment state changes, webhook processing, audit inserts, and short updates.
- REPEATABLE_READ: useful for stable calculations, payout previews, fee calculations, limit checks, and batch preparation where rereads must stay consistent.
- SERIALIZABLE: useful for strict balance reservation, last-seat style capacity, strict limits, or operations where concurrent transactions must behave as if executed one by one.

Best practices

- Do not choose isolation levels randomly.
- Start with the database default, but know what it is.
- Use stronger isolation only where the business case requires it.
- Keep high-isolation transactions short.
- Avoid remote API calls inside high-isolation transactions.
- Use optimistic locking when the real problem is concurrent updates.
- Use database constraints when the real problem is invalid state.
- Use idempotency keys for payment retries and duplicate requests.
- Use explicit locking only when needed.

Example optimistic locking:

```
@Entity
public class Account {

    @Id
    private Long id;

    @Version
    private Long version;

    private BigDecimal balance;
}
```

Optimistic locking helps detect concurrent updates without making every transaction overly restrictive.

A common mistake is using the isolation level as a magic fix. It is not.

If two users update the same entity at the same time, optimistic locking may be the better solution.

If two transactions must not create duplicate records, a unique constraint may be the better solution.

If payment retries can create duplicates, an idempotency key may be the better solution.

If a workflow spans multiple services, database isolation will not solve the whole problem.

Another common mistake is keeping a strong database transaction open while waiting for a remote system.

```
@Transactional(isolation = Isolation.SERIALIZABLE)
public void processPayment(Long paymentId) {
    Payment payment = paymentRepository.findById(paymentId).orElseThrow();

    paymentGateway.charge(payment); // slow external call

    payment.markPaid();
}
```

This can hurt the entire application.

A safer approach is to keep the database transaction short and process the external call separately.

```
@Transactional
public void markPaymentPending(Long paymentId) {
    Payment payment = paymentRepository.findById(paymentId).orElseThrow();
    payment.markPending();
}
```

Isolation level is not a technical decoration. It is a business consistency decision.

Ask these questions before changing it:

- What data must remain stable during this operation?
- What anomalies are acceptable?
- What is the cost of blocking other transactions?
- What does the database already guarantee?
- What should be solved by isolation, and what should be solved by locking, constraints, idempotency, or better workflow design?

The best isolation level is not always the strongest one. It is the weakest one that still protects the business rule.