Great! Let's go over **real-world use cases** for each of the following:

1. **Transaction Isolation Levels**
2. **Transaction Propagation Types**

Each has its importance depending on **concurrency** and **business logic** requirements.

**🔐 1. Transaction Isolation Levels – Use Cases**

| **Isolation Level** | **When to Use** | **Example Scenario** |
| --- | --- | --- |
| READ\_UNCOMMITTED | 🛑 Rarely used. For read-only operations where **dirty data is okay**. | Log aggregation, analytics systems where exact accuracy isn't critical |
| READ\_COMMITTED (default) | ✅ Commonly used when **dirty reads must be avoided**, but phantom rows are okay | Fetching user account or profile data safely |
| REPEATABLE\_READ | When multiple reads in the same transaction must return **consistent results** | Generating invoices: you don’t want the total to change during processing |
| SERIALIZABLE | When you must **completely isolate** concurrent transactions (safe but slow) | Online ticket booking / bank transfers – avoid race conditions |
| DEFAULT | Use database's default (usually READ\_COMMITTED) | Basic CRUD operations |

**🧭 2. Transaction Propagation – Use Cases**

**✅ REQUIRED (default)**

* Joins an existing transaction, or starts a new one if none exists
* ✅ Best for most service methods

**Use Case:**

java

CopyEdit

public void placeOrder() {

saveOrder(); // part of tx

deductStock(); // part of tx

}

If any step fails, the entire transaction rolls back.

**🔁 REQUIRES\_NEW**

* **Always creates a new transaction**, suspends the current one

**Use Case:**

🔒 You want to **log audit data** even if main transaction fails.

java

CopyEdit

@Transactional

public void updateUser() {

userRepo.save(); // main tx

logService.logActivity(); // runs in new tx

throw new RuntimeException(); // rolls back main tx, log is still saved

}

**🧩 NESTED**

* Creates a **nested savepoint** inside the existing transaction

**Use Case:**

✅ If nested method fails, rollback only that part, not the whole transaction.

java

CopyEdit

@Transactional

public void createInvoice() {

saveMain(); // part of outer tx

try {

saveDetails(); // nested tx

} catch (Exception e) {

// only saveDetails rolls back

}

}

⚠️ Only works with databases that support **savepoints** (like PostgreSQL)

**⚙️ SUPPORTS**

* Runs in current transaction **if exists**, otherwise **non-transactional**

**Use Case:**

Optional participation in a transaction.

java

CopyEdit

@Transactional

public void update() {

repo.save();

logger.log(); // SUPPORTS: may or may not be part of tx

}

**🚫 NOT\_SUPPORTED**

* Always runs **outside of a transaction**

**Use Case:**

You want a method to bypass the transaction system (e.g., read-only metrics).

java

CopyEdit

@Transactional

public void processData() {

repository.save(); // part of tx

metricsService.collect(); // NOT\_SUPPORTED: skips tx for performance

}

**❌ NEVER**

* Throws exception if a transaction already exists

**Use Case:**

Ensure method **only runs non-transactionally**

java

CopyEdit

@NEVER

public void scheduleCleanup() {

// must run outside tx, or fail

}

**🔒 MANDATORY**

* Must be called **inside a transaction**, or throws an exception

**Use Case:**

Enforce transactional context

java

CopyEdit

@MANDATORY

public void applyCoupon() {

// coupon logic must be part of the tx

}

**✅ Summary Table**

| **Propagation** | **Starts New TX?** | **Joins Existing TX?** | **Use Case** |
| --- | --- | --- | --- |
| REQUIRED | If none | ✅ Yes | Standard service methods |
| REQUIRES\_NEW | ✅ Always | ❌ No | Audit logs, notifications |
| NESTED | Nested | ✅ Yes | Partial rollback inside main transaction |
| SUPPORTS | ❌ No | ✅ If exists | Optional transactional methods |
| NOT\_SUPPORTED | ❌ No | ❌ No | Performance metrics, health checks |
| NEVER | ❌ No | ❌ Throws error | Ensure no transaction is active |
| MANDATORY | ❌ No | ✅ Required | Enforce method is called inside a transaction |

**👨‍💻 Real-world Example Scenario**

**Online Order Processing**

java

CopyEdit

@Transactional

public void placeOrder(Order order) {

orderRepository.save(order); // REQUIRED

try {

inventoryService.reduceStock(order); // REQUIRED

paymentService.charge(order); // REQUIRES\_NEW (log even if failure)

} catch (Exception ex) {

// payment failed → rollback order & inventory

}

notificationService.sendEmail(order); // NOT\_SUPPORTED

}

**@Transactional(rollbackFor = Exception.class)**

public void updateAccount() throws Exception {

// If any Exception (checked or unchecked) is thrown, rollback

}