Transaction Management

The proxy-based mechanism behind @Transactional, rollback rules, propagation (REQUIRED/REQUIRES_NEW), the self-invocation pitfall, isolation levels, readOnly, TransactionTemplate, and @TransactionalEventListener.

Advanced 55 min

Last updated: August 18, 2026
Read online: https://www.learnforgex.com/en/topics/transaction-management

---

Transaction Management

In the Spring Core lessons so far, we've seen how the container finds, defines, and configures beans. This final lesson tackles a different problem: how do we guarantee that several database operations either all happen together, or none of them happen at all? `@Transactional` is where the `TransactionManager` bean set up automatically in the Auto-Configuration & Properties lesson (see "This Project's Own application.yml and Config Classes") actually gets used -- this lesson covers what that bean does behind the scenes, when it actually matters, and where it's most often misunderstood.

What Is a Transaction?

Completely independent of Spring, as a plain database concept, a transaction turns several operations into one indivisible unit:

```
BEGIN
   ↓
UPDATE account_a
   ↓
UPDATE account_b
   ↓
COMMIT
```

If one of the operations fails, everything done up to that point is undone:

```
BEGIN
   ↓
UPDATE account_a
   ↓
(an error occurred)
   ↓
ROLLBACK
```

Four properties guarantee this behavior, known by the acronym ACID: **Atomicity** (either all the operations happen, or none do), **Consistency** (a transaction moves the database from one valid state to another valid state), **Isolation** (concurrently running transactions don't see each other's in-progress state, see "Isolation Levels (A Quick Look)"), and **Durability** (a committed transaction is permanent, even if the server crashes right afterward).

Why Does It Exist?

The classic example: transferring 100 units from account A to account B. There are two separate steps -- subtract from A, add to B. Without a transaction, if the second step fails for any reason (a network error, the application crashing, an exception):

```
A: -100
B: unchanged  # -- the money vanished
```

With a transaction, only one of two outcomes is possible:

```
A: -100, B: +100   (both succeeded)
```

or

```
A: unchanged, B: unchanged   (both rolled back)
```

The state "subtracted from A but never added to B" can **never** be observed from the outside -- the entire point of a transaction is to hide that in-between, inconsistent state completely.

History

Spring designed transaction management as a central part of the framework from the very beginning (2003, from its earliest pre-1.0 releases) -- at the time, J2EE's own transaction API (JTA) was both heavyweight and only worked inside an application server; Spring's `PlatformTransactionManager` abstraction let the exact same `@Transactional` code work, unchanged, across completely different underlying layers -- JDBC, Hibernate, JTA. Annotation-based `@Transactional` (replacing XML's `<tx:advice>` configuration) arrived in Spring 2.0 (2006) -- exactly the same XML-to-annotations transition we mentioned in the

Component Scanning lesson. `@EnableTransactionManagement` (XML-free setup with Java Config) was added in Spring 3.1 (2011). `@TransactionalEventListener` is newer still, arriving in Spring 4.2 (2015).

@Transactional: The Most Basic Use

For this lesson's `@Transactional` / `TransactionTemplate` examples to **genuinely** run and show real commit/rollback behavior, we use a hand-written, in-memory "ledger" (`Ledger`) instead of a real database, plus a tiny `PlatformTransactionManager` that manages it. There's no real Postgres connection available in this environment -- the same technique the Dependency Injection lesson used to hand-simulate a container applies here too. **Real projects never write a class like this** -- Spring Boot's own auto-configuration (`DataSourceAutoConfiguration` , `JpaTransactionManager`) sets this up for you (see the Auto-Configuration & Properties lesson):

```
import org.springframework.transaction.TransactionDefinition;
import org.springframework.transaction.support.AbstractPlatformTransactionManager;
import org.springframework.transaction.support.DefaultTransactionStatus;
import org.springframework.transaction.support.TransactionSynchronizationManager;

import java.util.ArrayList;
import java.util.List;

// A tiny in-memory "resource" standing in for a real database table, plus a
// custom PlatformTransactionManager that manages it. Real Spring Boot
// applications never write a class like this themselves -- Spring Boot's own
// auto-configuration registers a JpaTransactionManager/DataSourceTransactionManager
// for you (see the Auto-Configuration & Properties lesson). This project has no
// real database connection available in this environment, so instead we build
// the smallest possible *real* PlatformTransactionManager, purely so the
// examples in this lesson can genuinely run @Transactional/TransactionTemplate
// code with real commit/rollback/propagation behavior -- the same technique the
// Dependency Injection lesson used to hand-simulate a container before showing
// the real one.
//
// Writes made during an active transaction are buffered separately (never
// touching the committed state directly) and only merged in -- by appending,
// never by wholesale replacing -- when that specific transaction commits. This
// is what makes PROPAGATION_REQUIRES_NEW's independence actually work: a
// suspended outer transaction's own (still uncommitted) buffer is completely
// untouched by an inner transaction committing or rolling back, and vice versa.
class Ledger {
    private final List<String> committed = new ArrayList<>();

    void add(String entry) {
        currentBuffer().add(entry);
    }

    // Committed entries, plus whatever the CURRENT transaction (if any) has
    // written so far but not yet committed -- "read your own writes," the same
    // way a real database transaction sees its own uncommitted changes.
    List<String> entries() {
        List<String> visible = new ArrayList<>(committed);
```

```
        Object bound = TransactionSynchronizationManager.getResource(this);
        if (bound != null) {
            @SuppressWarnings("unchecked")
            List<String> buffer = (List<String>) bound;
            visible.addAll(buffer);
        }
        return List.copyOf(visible);
    }

    @SuppressWarnings("unchecked")
    private List<String> currentBuffer() {
        Object bound = TransactionSynchronizationManager.getResource(this);
        if (bound != null) {
            return (List<String>) bound;
        }
        // No active transaction -- write straight to the committed state.
        return committed;
    }

    // The following two methods are only ever called by LedgerTransactionManager.

    List<String> newBuffer() {
        return new ArrayList<>();
    }

    void applyBuffer(List<String> buffer) {
        committed.addAll(buffer);
    }
}

// Extending AbstractPlatformTransactionManager is the exact same extension
// point Spring's own DataSourceTransactionManager and JpaTransactionManager
// use -- we're just managing a plain in-memory list instead of a JDBC
// Connection/EntityManager. The pattern (bind a per-transaction buffer to the
// current thread via TransactionSynchronizationManager, keyed by the resource
// itself) mirrors how the real JDBC transaction manager tracks its Connection.
class LedgerTransactionManager extends AbstractPlatformTransactionManager {

    private final Ledger ledger;

    LedgerTransactionManager(Ledger ledger) {
        this.ledger = ledger;
    }

    // The "transaction object" -- created fresh for every getTransaction()
    // call, holding whatever this specific transaction needs to commit/detect
    // participation later.
    private static class LedgerTransaction {
        List<String> buffer;
        boolean newTransaction;
    }

    @Override
    protected Object doGetTransaction() {
        LedgerTransaction transaction = new LedgerTransaction();
        Object bound = TransactionSynchronizationManager.getResource(ledger);
        if (bound != null) {
            @SuppressWarnings("unchecked")
```

```
            List<String> existingBuffer = (List<String>) bound;
            transaction.buffer = existingBuffer;
        }
        return transaction;
    }

    @Override
    protected boolean isExistingTransaction(Object transaction) {
        // A thread already has a buffer bound -- REQUIRED will join it instead
        // of starting a new one.
        return ((LedgerTransaction) transaction).buffer != null;
    }

    @Override
    protected void doBegin(Object transactionObject, TransactionDefinition definition) {
        LedgerTransaction transaction = (LedgerTransaction) transactionObject;
        List<String> buffer = ledger.newBuffer();
        transaction.buffer = buffer;
        transaction.newTransaction = true;
        TransactionSynchronizationManager.bindResource(ledger, buffer);
    }

    @Override
    protected void doCommit(DefaultTransactionStatus status) {
        LedgerTransaction transaction = (LedgerTransaction) status.getTransaction();
        ledger.applyBuffer(transaction.buffer);
    }

    @Override
    protected void doRollback(DefaultTransactionStatus status) {
        // Nothing to do -- this transaction's buffer was never merged into the
        // committed state, so simply discarding it (in doCleanupAfterCompletion
        // below) is enough. Whatever other, independent transactions already
        // committed in the meantime (see PROPAGATION_REQUIRES_NEW) is untouched.
    }

    @Override
    protected void doCleanupAfterCompletion(Object transactionObject) {
        LedgerTransaction transaction = (LedgerTransaction) transactionObject;
        if (transaction.newTransaction) {
            TransactionSynchronizationManager.unbindResource(ledger);
        }
    }

    // REQUIRES_NEW needs to "suspend" whatever transaction is currently active
    // before starting a brand new one. Our resource is a single in-memory
    // object (not a pooled connection), so suspending is just unbinding the
    // current buffer -- there's no real external resource to detach.
    @Override
    protected Object doSuspend(Object transactionObject) {
        return TransactionSynchronizationManager.unbindResource(ledger);
    }

    @Override
    protected void doResume(Object transactionObject, Object suspendedResources) {
        TransactionSynchronizationManager.bindResource(ledger, suspendedResources);
    }
}
```

With this infrastructure in place, we can now run real `@Transactional` code:

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Service;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.transaction.annotation.Transactional;

// @Transactional's most basic use: a method that either fully succeeds
// (commit) or fully fails (rollback) -- no partial state is ever visible from
// the outside.
@Service
class AccountService {

    private final Ledger ledger;

    AccountService(Ledger ledger) {
        this.ledger = ledger;
    }

    @Transactional
    void transferSuccessfully(String from, String to, int amount) {
        ledger.add(from + " -" + amount);
        ledger.add(to + " +" + amount);
    }

    @Transactional
    void transferAndFail(String from, String to, int amount) {
        ledger.add(from + " -" + amount);
        ledger.add(to + " +" + amount);
        throw new IllegalStateException("Payment provider unreachable");
    }
}

@Configuration
@EnableTransactionManagement
class AccountConfig {

    @Bean
    Ledger ledger() {
        return new Ledger();
    }

    @Bean
    PlatformTransactionManager transactionManager(Ledger ledger) {
        return new LedgerTransactionManager(ledger);
    }

    @Bean
    AccountService accountService(Ledger ledger) {
        return new AccountService(ledger);
    }
}
```

```
class TransactionalBasicExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(AccountConfig.class);
        Ledger ledger = context.getBean(Ledger.class);

        context.getBean(AccountService.class).transferSuccessfully("A", "B", 100);
        System.out.println(ledger.entries());
        // [A -100, B +100]

        try {
            context.getBean(AccountService.class).transferAndFail("A", "B", 50);
        } catch (IllegalStateException e) {
            System.out.println("Failed: " + e.getMessage());
            // Failed: Payment provider unreachable
        }
        System.out.println(ledger.entries());
        // [A -100, B +100]  -- the failed transfer's two entries were rolled back

        context.close();
    }
}
```

`transferSuccessfully` completes both `ledger.add(...)` calls successfully and commits; `transferAndFail` makes the same two calls but then throws an exception -- both are rolled back, leaving no trace in the ledger.

Commit and Rollback Flow

For a successful method call, the flow works like this:

```
method starts
     ↓
transaction starts
     ↓
database operations
     ↓
method succeeds
     ↓
COMMIT
```

When an exception is thrown instead (which exception types trigger a rollback is the subject of the next section):

```
method starts
     ↓
transaction starts
     ↓
database operations
     ↓
exception is thrown
     ↓
ROLLBACK
```

This decision -- commit or rollback -- is made automatically by the proxy that handles `@Transactional`, right after the method returns (or throws) -- you never call `commit()` / `rollback()` by hand yourself (except with the programmatic approach, see "Programmatic Transactions: TransactionTemplate").

Rollback Rules: RuntimeException vs. Checked Exception

Here's a behavior that surprises most people: **Spring does not automatically roll back on every exception.** The default rule treats unchecked exceptions (`RuntimeException` and its subclasses, plus `Error`) as rollback triggers; checked exceptions (subclasses of `Exception` that aren't `RuntimeException`, e.g. `IOException`) **do not trigger a rollback** -- the transaction commits despite the exception:

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Service;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.transaction.annotation.Transactional;

import java.io.IOException;

// Spring's default rollback rule: unchecked exceptions (RuntimeException and
// its subclasses, plus Error) trigger a rollback; checked exceptions do NOT,
// unless you explicitly say otherwise with rollbackFor.
@Service
class ReportService {

    private final Ledger ledger;

    ReportService(Ledger ledger) {
        this.ledger = ledger;
    }

    @Transactional
    void writeThenThrowUnchecked() {
        ledger.add("report-draft");
        throw new IllegalStateException("Unchecked -- rolls back by default");
    }

    @Transactional
```

```
    void writeThenThrowChecked() throws IOException {
        ledger.add("report-draft");
        throw new IOException("Checked -- does NOT roll back by default");
    }

    @Transactional(rollbackFor = IOException.class)
    void writeThenThrowCheckedWithRollbackFor() throws IOException {
        ledger.add("report-draft");
        throw new IOException("Checked, but rollbackFor makes it roll back anyway");
    }
}

@Configuration
@EnableTransactionManagement
class ReportConfig {

    @Bean
    Ledger ledger() {
        return new Ledger();
    }

    @Bean
    PlatformTransactionManager transactionManager(Ledger ledger) {
        return new LedgerTransactionManager(ledger);
    }

    @Bean
    ReportService reportService(Ledger ledger) {
        return new ReportService(ledger);
    }
}

class RollbackRulesExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(ReportConfig.class);
        Ledger ledger = context.getBean(Ledger.class);
        ReportService service = context.getBean(ReportService.class);

        try {
            service.writeThenThrowUnchecked();
        } catch (IllegalStateException e) {
            System.out.println(ledger.entries());
            // []  -- rolled back, the unchecked exception undid the write
        }

        try {
            service.writeThenThrowChecked();
        } catch (IOException e) {
            System.out.println(ledger.entries());
            // [report-draft]  -- NOT rolled back, checked exceptions commit by
            // default even though an exception was thrown
        }

        try {
            service.writeThenThrowCheckedWithRollbackFor();
        } catch (IOException e) {
            System.out.println(ledger.entries());
```

```
            // [report-draft]  -- still just the one entry committed by the
            // previous call; rollbackFor rolled this call's own write back, so
            // it was never added a second time
        }

        context.close();
    }
}
```

`writeThenThrowUnchecked` gets rolled back because it throws `IllegalStateException` (unchecked). `writeThenThrowChecked` **commits** even though it throws `IOException` (checked) -- the written line becomes permanent, and the exception is only reported to the caller. `@Transactional(rollbackFor = IOException.class)` explicitly overrides this default, making even a checked exception trigger a rollback.

> This default rule comes from a historical distinction (checked exceptions were traditionally considered "expected, part of the normal flow," unchecked ones "unexpected failures"), but plenty of teams find it confusing today. If you write a `@Transactional` method that throws a checked exception and you want the data to stay consistent, never forget to add `rollbackFor` -- otherwise, code that catches the exception may not realize the data was never actually rolled back.

@EnableTransactionManagement and the Proxy-Based Mechanism

`@Transactional` doesn't resemble any of the mechanisms we saw in the Component Scanning and Spring IoC Container lessons -- it doesn't define a bean, and it doesn't state a condition either. Instead, while `@EnableTransactionManagement` is active, Spring wraps a **proxy** around every bean containing `@Transactional`:

```
Client
  ↓
Spring Proxy (TransactionInterceptor)
  ↓
transaction starts
  ↓
Target Method
  ↓
commit / rollback
```

This is an application of Spring AOP (Aspect-Oriented Programming) -- `TransactionInterceptor` is an "advice" that intercepts the method call before it ever reaches the real object. For classes found via component scanning (like `@Component` / `@Service`), the proxy is built with CGLIB (by subclassing) or a JDK dynamic proxy (if an interface exists) -- a real, concrete application of the `BeanPostProcessor` mechanism we saw in the Spring IoC Container lesson (see "The Bean Lifecycle: How the Container Builds a Bean") kicking in

exactly this way. The proxy's single most important consequence is the subject of the next section.

Self-Invocation Pitfall

A proxy can only intercept calls that come **through the bean** -- a call made via `this` (from inside the same class) never goes through the proxy at all, which means `@Transactional` is silently never applied:

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Service;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.transaction.annotation.Transactional;

// @Transactional works through a proxy Spring wraps around the bean (see
// "@EnableTransactionManagement and the Proxy-Based Mechanism"). Calling a
// @Transactional method through `this` (from inside the same class) never
// goes through that proxy at all -- the annotation is silently ignored.
@Service
class InvoiceService {

    private final Ledger ledger;

    InvoiceService(Ledger ledger) {
        this.ledger = ledger;
    }

    // Looks like it delegates to a transactional method -- but calling
    // writeLine(...) here is a plain Java method call on `this`, not a call
    // through the Spring proxy. @Transactional on writeLine() has NO effect
    // when reached this way.
    void createInvoiceViaSelfInvocation(String customer) {
        writeLine("invoice-open:" + customer);
        writeLine("invoice-line:" + customer);
        throw new IllegalStateException("Something went wrong after writing both lines");
    }

    @Transactional
    void writeLine(String line) {
        ledger.add(line);
    }
}

@Configuration
@EnableTransactionManagement
class InvoiceConfig {

    @Bean
    Ledger ledger() {
        return new Ledger();
    }
```

```
    @Bean
    PlatformTransactionManager transactionManager(Ledger ledger) {
        return new LedgerTransactionManager(ledger);
    }

    @Bean
    InvoiceService invoiceService(Ledger ledger) {
        return new InvoiceService(ledger);
    }
}

class SelfInvocationExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(InvoiceConfig.class);
        Ledger ledger = context.getBean(Ledger.class);

        try {
            context.getBean(InvoiceService.class).createInvoiceViaSelfInvocation("Ayse");
        } catch (IllegalStateException e) {
            System.out.println("Failed: " + e.getMessage());
        }

        // Both lines survive the exception -- writeLine(...) was never
        // actually transactional here, since it was called as
        // `this.writeLine(...)`, bypassing the proxy entirely. No transaction
        // ever started, so there was nothing to roll back.
        System.out.println(ledger.entries());
        // [invoice-open:Ayse, invoice-line:Ayse]

        context.close();
    }
}
```

`createInvoiceViaSelfInvocation` calls `writeLine(...)` as `this.writeLine(...)` -- not the proxy object you got from the container, but the real object directly. Even though `writeLine` is marked `@Transactional`, no transaction ever starts along this call path, so there's nothing to roll back either.

Propagation: REQUIRED (the Default)

`PROPAGATION_REQUIRED` **joins** an already-active transaction if one exists -- it doesn't start a second one. If the outer transaction rolls back, everything written by any `REQUIRED` method it called rolls back with it, because they were really all the same transaction all along:

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Service;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.transaction.annotation.Transactional;

// PROPAGATION_REQUIRED (the default): if a transaction is already active,
```

```
// join it -- don't start a second one. If the outer transaction rolls back,
// every write made by any REQUIRED method it called rolls back with it,
// because they were really all the same transaction all along.
@Service
class OrderService {

    private final PaymentService paymentService;
    private final Ledger ledger;

    OrderService(PaymentService paymentService, Ledger ledger) {
        this.paymentService = paymentService;
        this.ledger = ledger;
    }

    @Transactional
    void placeOrderThatFailsAfterPayment(String orderId) {
        ledger.add("order-created:" + orderId);
        paymentService.charge(orderId, 100); // joins this same transaction
        throw new IllegalStateException("Inventory check failed after payment");
    }
}

@Service
class PaymentService {
    private final Ledger ledger;

    PaymentService(Ledger ledger) {
        this.ledger = ledger;
    }

    @Transactional // PROPAGATION_REQUIRED is the default -- no explicit value needed
    void charge(String orderId, int amount) {
        ledger.add("payment-charged:" + orderId + ":" + amount);
    }
}

@Configuration
@EnableTransactionManagement
class OrderConfig {

    @Bean
    Ledger ledger() {
        return new Ledger();
    }

    @Bean
    PlatformTransactionManager transactionManager(Ledger ledger) {
        return new LedgerTransactionManager(ledger);
    }

    @Bean
    PaymentService paymentService(Ledger ledger) {
        return new PaymentService(ledger);
    }

    @Bean
    OrderService orderService(PaymentService paymentService, Ledger ledger) {
        return new OrderService(paymentService, ledger);
```

```
    }
}

class PropagationRequiredExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(OrderConfig.class);
        Ledger ledger = context.getBean(Ledger.class);

        try {
            context.getBean(OrderService.class).placeOrderThatFailsAfterPayment("ORD-1");
        } catch (IllegalStateException e) {
            System.out.println("Failed: " + e.getMessage());
        }

        // Both the order AND the payment are gone -- placeOrderThatFailsAfterPayment
        // and charge(...) shared the exact same transaction (PROPAGATION_REQUIRED
        // joined instead of starting a new one), so the whole thing rolled back
        // together.
        System.out.println(ledger.entries());
        // []

        context.close();
    }
}
```

`placeOrderThatFailsAfterPayment` and `charge(...)` share the same transaction -- `charge`'s own `@Transactional` doesn't start a new one, it joins the existing one. Even though the payment was successfully "written," when the order later fails, both are rolled back together.

Propagation: REQUIRES_NEW

`PROPAGATION_REQUIRES_NEW` **suspends** whatever transaction is active, even if one exists, and starts a completely independent, brand new one. This new transaction commits or rolls back entirely on its own -- if the outer transaction later rolls back, the inner one's already-committed work is untouched:

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Service;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

// PROPAGATION_REQUIRES_NEW: always suspend whatever transaction is active (if
// any) and start a brand new, completely independent one. The new transaction
// commits or rolls back entirely on its own -- if the *outer* transaction
// later rolls back, the inner one's already-committed work is untouched.
@Service
class AuditService {
    private final Ledger ledger;
```

```
    AuditService(Ledger ledger) {
        this.ledger = ledger;
    }

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    void recordAuditEntry(String message) {
        ledger.add("audit:" + message);
    }
}

@Service
class CheckoutService {
    private final AuditService auditService;
    private final Ledger ledger;

    CheckoutService(AuditService auditService, Ledger ledger) {
        this.auditService = auditService;
        this.ledger = ledger;
    }

    @Transactional
    void checkoutThatFails(String orderId) {
        ledger.add("checkout-started:" + orderId);
        auditService.recordAuditEntry("checkout attempted for " + orderId); // its own,
separate transaction
        throw new IllegalStateException("Card declined");
    }
}

@Configuration
@EnableTransactionManagement
class CheckoutConfig {

    @Bean
    Ledger ledger() {
        return new Ledger();
    }

    @Bean
    PlatformTransactionManager transactionManager(Ledger ledger) {
        return new LedgerTransactionManager(ledger);
    }

    @Bean
    AuditService auditService(Ledger ledger) {
        return new AuditService(ledger);
    }

    @Bean
    CheckoutService checkoutService(AuditService auditService, Ledger ledger) {
        return new CheckoutService(auditService, ledger);
    }
}

class PropagationRequiresNewExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(CheckoutConfig.class);
```

```
        Ledger ledger = context.getBean(Ledger.class);

        try {
            context.getBean(CheckoutService.class).checkoutThatFails("ORD-2");
        } catch (IllegalStateException e) {
            System.out.println("Failed: " + e.getMessage());
        }

        // "checkout-started" is gone (the outer transaction rolled back), but
        // the audit entry survives -- REQUIRES_NEW made it its own,
        // independent transaction that had already committed before
        // checkoutThatFails ever threw.
        System.out.println(ledger.entries());
        // [audit:checkout attempted for ORD-2]

        context.close();
    }
}
```

`checkoutThatFails` fails and gets rolled back, but `recordAuditEntry(...)` -- thanks to `REQUIRES_NEW` -- had already committed in its own, separate transaction. In the real world, this shows exactly why an audit record should be kept independent of the ordinary business operation: the "we tried this" information should persist even if the actual operation fails.

Other Propagation Types (A Quick Look)

The remaining five propagation types aren't used often enough to be worth demonstrating with a working example in this environment, but knowing what they do still matters: **`NESTED`** creates a savepoint inside the outer transaction -- the inner part can be rolled back while the outer part continues unaffected (this requires a real JDBC savepoint, which our `Ledger` doesn't support). **`SUPPORTS`** joins an active transaction if one exists, otherwise runs without one. **`MANDATORY`** **requires** an active transaction to already exist -- it throws an exception if none does. **`NOT_SUPPORTED`** suspends any active transaction and runs the method entirely without one. **`NEVER`** throws an exception if an active transaction exists at all -- a guarantee that "this method must never be called inside a transaction."

Isolation Levels (A Quick Look)

Isolation determines how much of another, concurrently running transaction's **not yet committed** changes a transaction can see -- since this requires genuinely concurrent transactions and a real database, there's no working code example in this lesson, but the three classic problems it solves are worth knowing: **Dirty Read** (reading a change that hasn't been committed yet -- if that change is later rolled back, the value you read never really existed), **Non-Repeatable Read** (reading the same row twice within the same transaction and getting different values, because another transaction committed in between), **Phantom Read** (running the same query twice and getting a different **number of rows**, because another transaction inserted or deleted rows in between). Isolation levels (`READ_UNCOMMITTED`, `READ_COMMITTED`, `REPEATABLE_READ`, `SERIALIZABLE`) prevent these three

problems one by one, at the cost of increasingly strict locking/version checking -- set with `@Transactional(isolation = ...)`.

Isolation in PostgreSQL

This project's database is PostgreSQL, and PostgreSQL's isolation behavior has two practical characteristics worth knowing. First: PostgreSQL's **default** isolation level is `READ_COMMITTED` (Spring/JPA's own default, `Isolation.DEFAULT`, inherits this too -- meaning this project, without customizing anything, already runs under `READ_COMMITTED`). Second, and less well known: PostgreSQL **doesn't actually support** `READ_UNCOMMITTED` -- even if you request it, the engine silently upgrades it to `READ_COMMITTED`; in other words, a dirty read can never happen in PostgreSQL, even opting in on purpose. `REPEATABLE_READ` and `SERIALIZABLE` are implemented in PostgreSQL with "snapshot isolation" (MVCC) rather than locking -- under `SERIALIZABLE`, if a conflict is detected, the transaction can fail with a serialization error right at commit time; application code needs to retry in that case.

readOnly = true: What It Does, What It Doesn't

`@Transactional(readOnly = true)` gives Spring, and the underlying JPA/Hibernate layer, a **hint** -- it is not an actual restriction:

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Service;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.transaction.annotation.Transactional;

import java.util.List;

// readOnly = true is a HINT, not an enforced restriction at the Spring level --
// real transaction managers (JpaTransactionManager, for example) use it to
// apply optimizations (like skipping Hibernate's dirty-checking flush), and
// some database drivers use it to route reads to a replica -- but nothing in
// the @Transactional contract itself stops a readOnly transaction from
// writing.
@Service
class ReportingService {
    private final Ledger ledger;

    ReportingService(Ledger ledger) {
        this.ledger = ledger;
    }

    @Transactional(readOnly = true)
    List<String> generateReport() {
        return ledger.entries();
    }

    // This compiles and runs just fine, even though it's marked readOnly --
```

```
    // our simple LedgerTransactionManager (like most real ones) does not
    // reject writes just because readOnly = true was set.
    @Transactional(readOnly = true)
    void generateReportAndSneakilyWrite() {
        ledger.add("this should not really happen, but nothing stops it");
    }
}

@Configuration
@EnableTransactionManagement
class ReportingConfig {

    @Bean
    Ledger ledger() {
        return new Ledger();
    }

    @Bean
    PlatformTransactionManager transactionManager(Ledger ledger) {
        return new LedgerTransactionManager(ledger);
    }

    @Bean
    ReportingService reportingService(Ledger ledger) {
        return new ReportingService(ledger);
    }
}

class ReadOnlyExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(ReportingConfig.class);
        ReportingService service = context.getBean(ReportingService.class);

        service.generateReportAndSneakilyWrite();
        System.out.println(service.generateReport());
        // [this should not really happen, but nothing stops it]

        context.close();
    }
}
```

`generateReportAndSneakilyWrite` writes without any trouble at all, despite being marked `readOnly = true` -- neither Spring's `@Transactional` contract nor our simple `LedgerTransactionManager` prevents it. In a real `JpaTransactionManager`, `readOnly = true`'s actual benefit is performance: it disables Hibernate's "dirty checking" (see "Spring Data JPA and Dirty Checking") mechanism and skips the flush, and some JDBC drivers use it to route reads to a replica. But it does **not** mean "this method definitely cannot write to the database" -- if you want that guarantee, you need to grant the database user itself read-only privileges.

Transaction Boundary: Why the Service Layer?

Where should a transaction start? In a typical layered architecture:

```
Controller
    ↓
Service   ← the transaction boundary belongs here
    ↓
Repository
```

Putting `@Transactional` on the **service** layer is the widely accepted rule, for two reasons: first, a single service method usually makes several repository calls (see the `OrderService` -> `PaymentService` example in "Propagation: REQUIRED (the Default)") -- drawing the transaction boundary here makes all of those calls one unit. Second, putting `@Transactional` on the controller layer (see "Common Mistakes") keeps the transaction unnecessarily broad -- work that has nothing to do with the database, like rendering a view or serializing JSON, stays inside the transaction too.

Programmatic Transactions: TransactionTemplate

`TransactionTemplate` is `@Transactional`'s programmatic counterpart -- for cases where the transaction boundary isn't "the whole method," or needs to be conditional:

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.support.TransactionTemplate;

// TransactionTemplate is @Transactional's programmatic counterpart -- useful
// when the transaction boundary needs to be conditional, or narrower than
// "the whole method," in a way an annotation on the method signature can't
// express.
@Configuration
class TransactionTemplateConfig {

    @Bean
    Ledger ledger() {
        return new Ledger();
    }

    @Bean
    PlatformTransactionManager transactionManager(Ledger ledger) {
        return new LedgerTransactionManager(ledger);
    }

    @Bean
    TransactionTemplate transactionTemplate(PlatformTransactionManager transactionManager) {
        return new TransactionTemplate(transactionManager);
    }
```

```
}

class TransactionTemplateExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(TransactionTemplateConfig.class);
        Ledger ledger = context.getBean(Ledger.class);
        TransactionTemplate transactionTemplate = context.getBean(TransactionTemplate.
class);

        // Runs the whole lambda inside a transaction -- commits normally if it
        // returns, rolls back if it throws, exactly like @Transactional.
        transactionTemplate.executeWithoutResult(status -> ledger.add("programmatic-
entry"));
        System.out.println(ledger.entries());
        // [programmatic-entry]

        try {
            transactionTemplate.executeWithoutResult(status -> {
                ledger.add("about-to-fail");
                throw new IllegalStateException("Rolled back programmatically");
            });
        } catch (IllegalStateException e) {
            System.out.println("Failed: " + e.getMessage());
        }
        System.out.println(ledger.entries());
        // [programmatic-entry]  -- "about-to-fail" was rolled back

        // status.setRollbackOnly() rolls back WITHOUT throwing an exception at
        // all -- useful when a business rule decides to abort, not an error.
        transactionTemplate.executeWithoutResult(status -> {
            ledger.add("conditionally-added");
            status.setRollbackOnly();
        });
        System.out.println(ledger.entries());
        // [programmatic-entry]  -- "conditionally-added" was rolled back too

        context.close();
    }
}
```

`executeWithoutResult` runs the entire lambda inside a transaction -- commits if it returns normally, rolls back if it throws, the same rule as `@Transactional`. `status.setRollbackOnly()` offers a different path: rolling back without throwing any exception at all, used when a business rule simply decides to abort.

Spring Data JPA and Dirty Checking

Spring Data JPA's own repository methods (`save()`, `findById()`, `delete()`, and so on -- interfaces like `TopicRepository`, generated as a proxy without ever writing `@Repository`, that we saw in the Component Scanning lesson's "This Project's Own Classes: A Real Component Scanning Example") are already `@Transactional` themselves (defined on `SimpleJpaRepository`). Beyond that, Hibernate's **dirty checking** feature means that when a

field of an entity managed within a transaction is modified, that change is written to the database at commit time even if `save()` is never called at all. Since this project is entirely read-only (see "This Project's Own Repositories: Why Is There Still No @Transactional?"), there's no real example of this, but hypothetically: if a `TopicService.updateDifficulty(slug, newDifficulty)` method existed, and inside it we called `topic.setDifficulty(newDifficulty)` on a managed `Topic` obtained via `topicRepository.findBySlug(slug)`, then even **without ever calling** `topicRepository.save(topic)`, Hibernate would notice the change and send an `UPDATE` query once the transaction committed.

> Dirty checking is disabled in `readOnly = true` transactions (see "readOnly = true: What It Does, What It Doesn't") -- Hibernate gains performance by skipping the pre-flush comparison for entities it knows will never be written.

Lazy Loading and LazyInitializationException

This project's `Topic`, `Category`, `TopicTranslation`, and `CodeExample` entities all have real `@ManyToOne(fetch = FetchType.LAZY)` relationships (`TopicTranslation.topic`, `CodeExample.topic`, `Category.course`, `Topic.category`). A lazy relationship is only fetched from the database when it's **explicitly accessed** (e.g. `topic.getCategory()`) -- and that access has to happen while the entity's persistence context (the Hibernate session) is still open. Trying to access it afterward throws `LazyInitializationException`.

`TopicRepository`'s real source code has a method that avoids exactly this problem:

```
@Query("select t from Topic t join fetch t.category c join fetch c.course where t.slug = :
slug")
Optional<Topic> findBySlugWithCategoryAndCourse(String slug);
```

`TopicController.show(...)` deliberately uses this instead of the plain `findBySlug(slug)` -- `join fetch` resolves the `category` and `course` relationships **in the same query**, right away, with no lazy loading involved. The comment in the source code says this explicitly: the breadcrumb and previous/next topic navigation need these relationships, and the project "resolves this explicitly in a single query instead of leaving it to lazy loading (open-in-view)." `spring.jpa.open-in-view` is never set in this project, so Spring Boot's default (`true`) applies -- meaning `findBySlug(slug)` would likely have still worked even if the Thymeleaf template accessed `topic.category.course.name` (open-in-view keeps the persistence context open until the view finishes rendering), but that's a widely recognized anti-pattern that holds a database connection open far longer than it needs to -- the project's `join fetch` choice avoids exactly that.

Transactional Events: @TransactionalEventListener and AFTER_COMMIT

We saw `ApplicationEvent` / `@EventListener` in the Auto-Configuration & Properties lesson -- `@TransactionalEventListener` makes the same idea transaction-aware: it handles an event not the moment it's published, but once the transaction reaches a particular phase. The most commonly used phase is `AFTER_COMMIT`:

```
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Component;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.transaction.event.TransactionPhase;
import org.springframework.transaction.event.TransactionalEventListener;

// @TransactionalEventListener (from the Auto-Configuration & Properties
// lesson's ApplicationEvent/@EventListener section, but transaction-aware)
// defers handling an event until the surrounding transaction reaches a
// particular phase -- most commonly AFTER_COMMIT, so a listener only reacts
// to writes that actually stuck.
class OrderCreatedEvent {
    private final String orderId;

    OrderCreatedEvent(String orderId) {
        this.orderId = orderId;
    }

    String getOrderId() {
        return orderId;
    }
}

@Component
class OrderCreationService {
    private final Ledger ledger;
    private final ApplicationEventPublisher publisher;

    OrderCreationService(Ledger ledger, ApplicationEventPublisher publisher) {
        this.ledger = ledger;
        this.publisher = publisher;
    }

    @Transactional
    void createOrder(String orderId, boolean simulateFailureAfterPublish) {
        ledger.add("order-created:" + orderId);
        publisher.publishEvent(new OrderCreatedEvent(orderId));
        if (simulateFailureAfterPublish) {
            throw new IllegalStateException("Something failed after publishing the event");
        }
```

```
    }
}

@Component
class OrderNotificationListener {
    // Only runs if the transaction that published the event actually commits --
    // if createOrder(...) rolls back, this method never runs at all, even
    // though publishEvent(...) was called.
    @TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
    void onOrderCreated(OrderCreatedEvent event) {
        System.out.println("Notification sent for order " + event.getOrderId());
    }
}

@Configuration
@EnableTransactionManagement
@ComponentScan
class OrderCreationConfig {

    @Bean
    Ledger ledger() {
        return new Ledger();
    }

    @Bean
    PlatformTransactionManager transactionManager(Ledger ledger) {
        return new LedgerTransactionManager(ledger);
    }
}

class TransactionalEventListenerExample {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(OrderCreationConfig.class);
        OrderCreationService service = context.getBean(OrderCreationService.class);

        service.createOrder("ORD-3", false);
        // Notification sent for order ORD-3

        try {
            service.createOrder("ORD-4", true);
        } catch (IllegalStateException e) {
            // No "Notification sent for order ORD-4" is ever printed -- the
            // transaction rolled back, so AFTER_COMMIT never fires.
            System.out.println("Failed: " + e.getMessage());
        }

        context.close();
    }
}
```

`createOrder` commits successfully and the listener runs when `simulateFailureAfterPublish = false`. When it's `true`, even though the event was published, the listener for `AFTER_COMMIT` **never runs at all**, because the transaction never committed -- the difference between publishing an event and that event's effect actually happening is clearly visible here.

Testing Transactions (A Quick Look)

Spring Test (`spring-boot-starter-test` , a dependency of this project) adds special behavior when a test method is marked `@Transactional` : the test method runs inside its own transaction, and that transaction is **automatically rolled back once the test finishes** -- by default, nothing the test writes to the database is ever permanent, and the next test starts with a clean database:

```
@SpringBootTest
class OrderServiceTest {

    @Test
    @Transactional
    void shouldCreateOrder() {
        // ... code that writes to the database ...
        // automatic rollback when the test finishes, no manual cleanup needed
    }
}
```

This is provided by `TransactionalTestExecutionListener` and requires a real Spring Boot test environment (`@SpringBootTest`) -- since that's a different execution model than the plain `AnnotationConfigApplicationContext` + `main()` shape this lesson's other examples use, there's no separate code example here. One thing worth watching for: if the code under test itself uses `REQUIRES_NEW` (see "Propagation: REQUIRES_NEW"), that inner transaction **genuinely commits**, independent of the test's own transaction -- the test's outer rollback cannot undo it.

This Project's Own Repositories: Why Is There Still No @Transactional?

There is no `@Transactional` anywhere in this project -- `grep -rn "@Transactional" src/main/java` comes back empty. The reason is simple: this project's real classes, like `NavigationService` , `ContentResolver` , and `TopicController` , are all **read-only** -- single, one-query repository calls like `courseRepository.findAll()` or `topicRepository.findBySlugWithCategoryAndCourse(...)` . Spring Data JPA's own `SimpleJpaRepository` (see "Spring Data JPA and Dirty Checking") already wraps every repository method in its own transaction -- adding `@Transactional` to the service layer on top of a single query would have provided no benefit at all.

The only "writes" this project ever sees don't come from the running application itself, they come from Flyway migrations (see `db/migration/`) -- every `INSERT` / `UPDATE` runs as plain SQL every time the application starts, with the service layer never involved at all. If this project ever gained an "admin panel" or a content-editing feature (say, a `TopicService.reorder(...)` method that changes a topic's `sort_order`), that's exactly when a real `@Transactional` service method would be needed -- likely updating several `Topic` rows as one unit, in a shape very similar to the `OrderService` example in "Propagation: REQUIRED (the Default)".

Best Practices

- **Put `@Transactional` on the service layer, not the controller** -- this keeps the transaction boundary limited to work that's actually about the database (see "Transaction Boundary: Why the Service Layer?").
- **Don't forget `rollbackFor` on a `@Transactional` method that throws a checked exception** -- the default behavior commits on checked exceptions, which usually isn't what you want (see "Rollback Rules: RuntimeException vs. Checked Exception").
- **Mark read-only methods `readOnly = true`** -- even though it's not an actual restriction, it improves performance under a real `JpaTransactionManager` (see "readOnly = true: What It Does, What It Doesn't").
- **Keep transactions short, and don't make an external API call inside one** -- if a payment provider or email service call is slow or fails, it holds the database connection (and any locks) open far longer than necessary; for work like that, `@TransactionalEventListener(phase = AFTER_COMMIT)` is a better fit (see "Transactional Events: @TransactionalEventListener and AFTER_COMMIT").
- **Don't use `REQUIRES_NEW` unnecessarily** -- every `REQUIRES_NEW` call means a separate transaction (and, on a real database, a separate connection); reserve it for work that genuinely needs to be independent of the outer transaction, like an audit record (see "Propagation: REQUIRES_NEW").

Common Mistakes

1. Putting `@Transactional` on the controller. The transaction boundary then covers work that has nothing to do with the database, like rendering a view -- the service layer is the right place (see "Transaction Boundary: Why the Service Layer?").

2. Assuming `@Transactional` will work through self-invocation. A call made via `this` never goes through the proxy at all -- the annotation silently does nothing (see "Self-Invocation Pitfall").

3. Assuming a method that throws a checked exception rolls back automatically. The default behavior is the opposite: checked exceptions allow a commit, unless `rollbackFor` is explicitly written (see "Rollback Rules: RuntimeException vs. Checked Exception").

4. Assuming `readOnly = true` guarantees "this method can't write." It's only a performance hint, not a restriction (see "readOnly = true: What It Does, What It Doesn't").

5. Changing the isolation level without fully understanding what it does. A stricter level (like `SERIALIZABLE`) prevents concurrency problems but reduces performance and can cause serialization failures (especially in PostgreSQL) -- clarify which problem (dirty read, non-repeatable read, phantom read) you're actually trying to solve before changing the default (see "Isolation Levels (A Quick Look)").

6. Trying to "solve" the lazy loading problem by spreading `@Transactional` / open-in-view everywhere. This keeps the database connection open far longer than necessary -- the correct fix is to explicitly fetch the relationships you need in a single query with `join fetch`, exactly like this project's `findBySlugWithCategoryAndCourse` does (see "Lazy Loading and LazyInitializationException").

Summary, Cheat Sheet, and Glossary

Transaction management turns several database operations into one indivisible unit -- `@Transactional` achieves this through a proxy that wraps the method call and decides to commit or roll back based on whether an exception was thrown. Key points:

- ACID: Atomicity, Consistency, Isolation, Durability
- Default rollback rule: unchecked exceptions (`RuntimeException` / `Error`) trigger a rollback, checked exceptions don't -- overridden with `rollbackFor`
- `@Transactional` works through a proxy (AOP, `TransactionInterceptor`) -- self-invocation (a call via `this`) bypasses that proxy
- Propagation: `REQUIRED` (the default, joins an existing one), `REQUIRES_NEW` (suspends and starts an independent new one), `NESTED` / `SUPPORTS` / `MANDATORY` / `NOT_SUPPORTED` / `NEVER` (the other, less commonly used types)
- Isolation levels prevent dirty read/non-repeatable read/phantom read problems, one by one, with increasing strictness; PostgreSQL's default is `READ_COMMITTED`
- `readOnly = true`: a performance hint, not a restriction
- `TransactionTemplate`: `@Transactional`'s programmatic counterpart
- `@TransactionalEventListener(phase = AFTER_COMMIT)`: handles an event only if the transaction that published it actually commits

Quick reference:

```
@Transactional                                      // REQUIRED, rolls back on all exceptions
except checked ones by default
@Transactional(rollbackFor = Exception.class)      // make checked exceptions trigger
rollback too
@Transactional(readOnly = true)                     // performance hint, not a restriction
@Transactional(propagation = Propagation.REQUIRES_NEW)  // always a new, independent
transaction
@Transactional(isolation = Isolation.SERIALIZABLE) // the strictest isolation

class MyService {
    // SELF-INVOCATION WARNING: this.otherMethod() bypasses the proxy.
    void outer() {
        this.inner(); // @Transactional is NOT applied, even though it's annotated
    }

    @Transactional
    void inner() { }
}

// Programmatic alternative:
transactionTemplate.executeWithoutResult(status -> {
    // ...
    if (someCondition) {
        status.setRollbackOnly(); // rolls back without throwing an exception
    }
});

@TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
void onSomeEvent(SomeEvent event) { }
```

Glossary

Transaction — An indivisible unit that guarantees several operations either all happen together (commit) or not at all (rollback).

ACID — Atomicity, Consistency, Isolation, Durability: the four properties a transaction must provide.

`@Transactional` — The annotation that wraps a method (or class) with a transaction boundary.

`PlatformTransactionManager` — Spring's interface abstracting starting/committing/rolling back a transaction; has real implementations like `DataSourceTransactionManager` and `JpaTransactionManager`.

Rollback rule — The rule that determines which exception types trigger a rollback; by default, only unchecked exceptions.

Self-invocation — A proxied bean calling its own method via `this`; because it bypasses the proxy, it disables proxy-based annotations like `@Transactional`.

Propagation — The setting that determines how a `@Transactional` method behaves when a transaction is already active (`REQUIRED`, `REQUIRES_NEW`, etc.).

Isolation — The setting that determines how much of a concurrently running transaction's uncommitted changes another transaction can see.

`readOnly` — A performance-oriented hint indicating a transaction will only read; not a restriction.

`TransactionTemplate` — `@Transactional`'s programmatic (annotation-free) counterpart.

`@TransactionalEventListener` — A listener annotation that handles an event once the transaction that published it reaches a particular phase (most commonly `AFTER_COMMIT`).

Appendix: Mini Project — A Money Transfer

This mini project completes the account-transfer scenario used throughout the lesson, bringing together rollback rules, `PROPAGATION_REQUIRED`, and the self-invocation pitfall:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Service;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.transaction.annotation.Transactional;

// Mini project: a small money-transfer service, reusing the same
// Ledger/LedgerTransactionManager infrastructure from earlier in this lesson,
// that ties together rollback rules, PROPAGATION_REQUIRED, and the
// self-invocation pitfall all at once -- the same account-transfer scenario
// used throughout this lesson, brought together into one realistic flow.
class InsufficientFundsException extends RuntimeException {
    InsufficientFundsException(String message) {
        super(message);
    }
}

@Service
class AccountRepository {
    private final Ledger ledger;

    AccountRepository(Ledger ledger) {
        this.ledger = ledger;
        // Opening balances, recorded as ledger entries just like any other change.
        ledger.add("A:+500");
        ledger.add("B:+100");
    }

    int balanceOf(String account) {
        int balance = 0;
        for (String entry : ledger.entries()) {
            String[] parts = entry.split(":");
            if (parts[0].equals(account)) {
                balance += Integer.parseInt(parts[1]);
            }
        }
        return balance;
```

```
    }

    @Transactional
    void debit(String account, int amount) {
        if (balanceOf(account) < amount) {
            // Unchecked -- rolls back automatically, no rollbackFor needed
            // (see "Rollback Rules" earlier in this lesson).
            throw new InsufficientFundsException(account + " has insufficient funds");
        }
        ledger.add(account + ":-" + amount);
    }

    @Transactional
    void credit(String account, int amount) {
        ledger.add(account + ":+" + amount);
    }
}

@Service
class MoneyTransferService {
    private final AccountRepository accountRepository;

    MoneyTransferService(AccountRepository accountRepository) {
        this.accountRepository = accountRepository;
    }

    // Both debit(...) and credit(...) are PROPAGATION_REQUIRED (the default),
    // so they join this same transaction -- if credit(...) fails, the debit
    // that already ran joins the rollback too, instead of leaving money
    // vanished from account A.
    @Transactional
    void transfer(String from, String to, int amount) {
        accountRepository.debit(from, amount);
        accountRepository.credit(to, amount);
    }

    // Deliberately buggy: transferViaSelfInvocation is NOT @Transactional
    // itself, and calls transferInternal(...) through `this` -- exactly the
    // self-invocation pitfall from earlier in this lesson. @Transactional on
    // transferInternal has no effect when reached this way.
    void transferViaSelfInvocation(String from, String to, int amount) {
        transferInternal(from, to, amount);
    }

    @Transactional
    void transferInternal(String from, String to, int amount) {
        accountRepository.debit(from, amount);
        accountRepository.credit(to, amount);
        throw new IllegalStateException("Simulated failure after both writes");
    }
}

@Configuration
@EnableTransactionManagement
@ComponentScan
class MoneyTransferConfig {

    @Bean
```

```
    Ledger ledger() {
        return new Ledger();
    }

    @Bean
    PlatformTransactionManager transactionManager(Ledger ledger) {
        return new LedgerTransactionManager(ledger);
    }
}
```

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

class MoneyTransferDemo {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(MoneyTransferConfig.class);
        AccountRepository accounts = context.getBean(AccountRepository.class);
        MoneyTransferService transferService = context.getBean(MoneyTransferService.class);

        System.out.println(accounts.balanceOf("A") + " / " + accounts.balanceOf("B"));
        // 500 / 100

        transferService.transfer("A", "B", 200);
        System.out.println(accounts.balanceOf("A") + " / " + accounts.balanceOf("B"));
        // 300 / 300

        try {
            transferService.transfer("A", "B", 10_000);
        } catch (InsufficientFundsException e) {
            System.out.println("Failed: " + e.getMessage());
        }
        System.out.println(accounts.balanceOf("A") + " / " + accounts.balanceOf("B"));
        // 300 / 300  -- unchanged, debit(...) itself rolled back before credit
        // ever ran

        try {
            transferService.transferViaSelfInvocation("A", "B", 50);
        } catch (IllegalStateException e) {
            System.out.println("Failed: " + e.getMessage());
        }
        // Both debit and credit survive -- self-invocation meant
        // transferInternal's own @Transactional never actually applied (it was
        // called via `this`), so there was no outer transaction to roll back.
        // debit(...) and credit(...) still ran as their own, separate,
        // already-committed transactions (they were called on the injected
        // accountRepository bean, a real proxy, not via `this`).
        System.out.println(accounts.balanceOf("A") + " / " + accounts.balanceOf("B"));
        // 250 / 350

        context.close();
    }
}
```

`transfer(...)` relies on `debit(...)` and `credit(...)` sharing the same transaction thanks to `PROPAGATION_REQUIRED` -- when a withdrawal is attempted from an account with insufficient

funds, `debit(...)` throws its own `InsufficientFundsException` (unchecked), and everything is rolled back. `transferViaSelfInvocation(...)` is deliberately broken: because it calls `transferInternal(...)` via `this`, that method's `@Transactional` never actually applies -- but `debit(...)` / `credit(...)`'s own `@Transactional` annotations (since they're called through `accountRepository`, a separate bean) work normally, each committing on its own.

Appendix: Mini Project — Order Processing

The final mini project brings propagation and transactional events together in a realistic `OrderService` -> `PaymentService` -> `InventoryAuditService` flow:

```
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.stereotype.Service;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.transaction.event.TransactionPhase;
import org.springframework.transaction.event.TransactionalEventListener;

// Mini project: a realistic OrderService -> PaymentService -> InventoryAuditService
// flow, tying propagation and transactional events together. The order and
// payment share one transaction (PROPAGATION_REQUIRED); the audit log is
// PROPAGATION_REQUIRES_NEW, so it survives even if the order later fails; and
// the shipping notification only fires once the whole order transaction has
// actually committed.
class OrderPlacedEvent {
    private final String orderId;

    OrderPlacedEvent(String orderId) {
        this.orderId = orderId;
    }

    String getOrderId() {
        return orderId;
    }
}

@Service
class InventoryAuditService {
    private final Ledger ledger;

    InventoryAuditService(Ledger ledger) {
        this.ledger = ledger;
    }

    // Independent of the order transaction on purpose -- an audit trail
    // should record "we tried this" even if the order itself is later rolled
    // back.
    @Transactional(propagation = Propagation.REQUIRES_NEW)
    void recordAttempt(String orderId) {
```

```
        ledger.add("audit:attempted:" + orderId);
    }
}

@Service
class PaymentService {
    private final Ledger ledger;

    PaymentService(Ledger ledger) {
        this.ledger = ledger;
    }

    @Transactional // PROPAGATION_REQUIRED -- joins the order's own transaction
    void charge(String orderId, int amount) {
        ledger.add("payment:" + orderId + ":" + amount);
    }
}

@Service
class OrderService {
    private final PaymentService paymentService;
    private final InventoryAuditService auditService;
    private final Ledger ledger;
    private final ApplicationEventPublisher publisher;

    OrderService(PaymentService paymentService,
                 InventoryAuditService auditService,
                 Ledger ledger,
                 ApplicationEventPublisher publisher) {
        this.paymentService = paymentService;
        this.auditService = auditService;
        this.ledger = ledger;
        this.publisher = publisher;
    }

    @Transactional
    void placeOrder(String orderId, int amount, boolean outOfStock) {
        auditService.recordAttempt(orderId); // its own transaction, always commits
        ledger.add("order:" + orderId);
        paymentService.charge(orderId, amount); // joins this transaction
        if (outOfStock) {
            throw new IllegalStateException("Out of stock -- order rolled back");
        }
        publisher.publishEvent(new OrderPlacedEvent(orderId));
    }
}

@Service
class ShippingNotificationListener {
    // Only fires once placeOrder(...) has actually committed -- never for the
    // out-of-stock case, since that transaction rolled back before the
    // event's transaction could ever reach AFTER_COMMIT.
    @TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT)
    void onOrderPlaced(OrderPlacedEvent event) {
        System.out.println("Shipping notification sent for order " + event.getOrderId());
    }
}
```

```
@Configuration
@EnableTransactionManagement
@ComponentScan
class OrderProcessingConfig {

    @Bean
    Ledger ledger() {
        return new Ledger();
    }

    @Bean
    PlatformTransactionManager transactionManager(Ledger ledger) {
        return new LedgerTransactionManager(ledger);
    }
}
```

```
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

class OrderProcessingDemo {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext
(OrderProcessingConfig.class);
        Ledger ledger = context.getBean(Ledger.class);
        OrderService orderService = context.getBean(OrderService.class);

        orderService.placeOrder("ORD-10", 250, false);
        // Shipping notification sent for order ORD-10
        System.out.println(ledger.entries());
        // [audit:attempted:ORD-10, order:ORD-10, payment:ORD-10:250]

        try {
            orderService.placeOrder("ORD-11", 90, true);
        } catch (IllegalStateException e) {
            System.out.println("Failed: " + e.getMessage());
        }
        // No "Shipping notification..." line for ORD-11 -- its transaction
        // rolled back before AFTER_COMMIT could ever fire.
        System.out.println(ledger.entries());
        // [audit:attempted:ORD-10, order:ORD-10, payment:ORD-10:250, audit:attempted:ORD-
11]
        // -- the audit entry for ORD-11 survives (REQUIRES_NEW, its own
        // transaction), but "order:ORD-11" and "payment:ORD-11:90" are gone.

        context.close();
    }
}
```

The order and payment share the same (`REQUIRED`) transaction, while the audit record (`recordAttempt`) is deliberately `REQUIRES_NEW` -- even if the order is later rolled back due to insufficient stock, the "we attempted this order" information stays permanent. The shipping notification only fires on `AFTER_COMMIT` -- when stock is insufficient, no notification is ever sent, because that transaction never commits.

> Every call to `InventoryAuditService.recordAttempt(...)` opens a separate connection /transaction on a real database -- as noted in "Best Practices," using `REQUIRES_NEW` only for work that genuinely needs to be independent matters for both performance and avoiding connection pool exhaustion.