Transaction Management

@Transactional'ın proxy tabanlı mekanizması, rollback kuralları, propagation (REQUIRED /REQUIRES_NEW), self-invocation tuzağı, isolation levels, readOnly, TransactionTemplate ve @TransactionalEventListener.

İleri 55 dk

Son güncelleme: 18 Ağustos 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/transaction-management

---

Transaction Management

Şimdiye kadarki Spring Core derslerinde container'ın bean'leri nasıl bulduğunu, tanımladığını ve yapılandırdığını gördük. Bu son ders farklı bir soruna eğiliyor: birden fazla veritabanı işleminin ya hep birlikte ya da hiç gerçekleşmemesini nasıl garanti ederiz? `@Transactional`, Auto-Configuration & Properties dersinde otomatik kurulan `TransactionManager` bean'ini (bkz. "Bu Projenin Kendi application.yml ve Config Sınıfları") gerçekten kullanan yer -- bu ders, o bean'in perde arkasında ne yaptığını, ne zaman işe yaradığını ve nerelerde yanlış anlaşıldığını işliyor.

Transaction Nedir?

Spring'den tamamen bağımsız, saf veritabanı kavramı olarak bir transaction, birden fazla işlemi tek, bölünemez bir birim hâline getirir:

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

İşlemlerden biri başarısız olursa, o ana kadar yapılan her şey geri alınır:

```
BEGIN
   ↓
UPDATE account_a
   ↓
(hata oluştu)
   ↓
ROLLBACK
```

Bu davranışı garanti eden dört özellik, ACID kısaltmasıyla anılır: **Atomicity** (bölünemezlik -- ya tüm işlemler ya hiçbiri), **Consistency** (tutarlılık -- transaction, veritabanını bir geçerli durumdan başka bir geçerli duruma taşır), **Isolation** (izolasyon -- aynı anda çalışan transaction'lar birbirinin ara durumunu görmez, bkz. "Isolation Levels (Kısa Bakış)") ve **Durability** (kalıcılık -- commit olan bir transaction, sunucu çökse bile kalıcıdır).

Neden Var?

Klasik örnek: A hesabından B hesabına 100 birim transfer. İki ayrı adım var -- A'dan düş, B'ye ekle. Transaction olmadan, ikinci adım herhangi bir sebeple (ağ hatası, uygulama çökmesi, bir exception) başarısız olursa:

```
A: -100
B: değişmedi  # -- para ortadan kayboldu
```

Transaction ile, yalnızca iki sonuçtan biri mümkündür:

```
A: -100, B: +100   (ikisi de başarılı)
```

veya

```
A: değişmedi, B: değişmedi   (ikisi de geri alındı)
```

"A'dan düşüldü ama B'ye hiç eklenmedi" durumu **hiçbir zaman** gözlemlenemez -- transaction'ın tüm amacı bu ara, tutarsız durumu dış dünyadan tamamen gizlemek.

Tarihçe

Spring, transaction yönetimini en başından beri (2003, Spring 1.0'dan önceki ilk sürümlerden itibaren) framework'ün merkezi bir parçası olarak tasarladı -- o dönemde J2EE'nin kendi transaction API'si (JTA) hem ağır hem de yalnızca uygulama sunucusu içinde çalışıyordu; Spring'in `PlatformTransactionManager` soyutlaması, aynı `@Transactional` kodunun JDBC, Hibernate, JTA gibi tamamen farklı alt katmanlarla, kod değişmeden çalışabilmesini sağladı. Annotation tabanlı `@Transactional` (XML'deki `<tx:advice>` yapılandırmasının yerini alarak) Spring 2.0'da (2006) geldi -- tam olarak Component Scanning dersinde bahsettiğimiz, XML'den annotation'lara geçiş döneminin bir parçası.

`@EnableTransactionManagement` (Java Config ile XML-siz kurulum) Spring 3.1'de (2011) eklendi. `@TransactionalEventListener` ise daha yeni, Spring 4.2'de (2015) geldi.

@Transactional: En Basit Kullanım

Bu derste `@Transactional` / `TransactionTemplate` örneklerinin **gerçekten** çalışıp commit /rollback davranışını göstermesi için, gerçek bir veritabanı yerine, elle yazılmış, in-memory bir "defter" (`Ledger`) ve onu yöneten minik bir `PlatformTransactionManager` kullanıyoruz. Bu ortamda gerçek bir Postgres bağlantısı yok; Dependency Injection dersinde container'ı elle simüle ettiğimiz teknik burada da geçerli. **Gerçek projelerde bu sınıf hiç yazılmaz** -- Spring Boot'un kendi auto-configuration'ı (`DataSourceAutoConfiguration`, `JpaTransactionManager`) bunu senin yerine kurar (bkz. Auto-Configuration & Properties dersi):

```
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);
    }
}
```

Bu altyapıyla artık gerçek `@Transactional` kodu çalıştırabiliriz:

```
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` her iki `ledger.add(...)` çağrısını da başarıyla tamamlıyor ve commit ediyor; `transferAndFail` aynı iki çağrıyı yapıyor ama sonra bir exception fırlatıyor -- ikisi de geri alınıyor, defterde hiçbir iz kalmıyor.

Commit ve Rollback Akışı

Başarılı bir metot çağrısında akış şöyle işler:

```
metot başlar
     ↓
transaction başlar
     ↓
veritabanı işlemleri
     ↓
metot başarıyla biter
     ↓
COMMIT
```

Bir exception fırlatıldığında ise (hangi exception'ların rollback tetiklediği bir sonraki bölümün konusu):

```
metot başlar
     ↓
transaction başlar
     ↓
veritabanı işlemleri
     ↓
exception fırlatılır
     ↓
ROLLBACK
```

Bu karar -- commit mi, rollback mi -- `@Transactional`'ı işleten proxy tarafından, metot geri döndükten (ya da exception fırlattıktan) hemen sonra, otomatik olarak verilir; sen hiçbir zaman elle `commit()` / `rollback()` çağırmazsın (programatik yöntem hariç, bkz. "Programmatic Transactions: TransactionTemplate").

Rollback Kuralları: RuntimeException vs Checked Exception

Burada çoğu kişinin şaşırdığı bir davranış var: **Spring her exception'da otomatik rollback yapmaz.** Varsayılan kural, unchecked exception'ları (`RuntimeException` ve alt sınıfları, artı `Error`) rollback tetikleyici sayar; checked exception'lar (`Exception`'ın `RuntimeException` olmayan alt sınıfları, örn. `IOException`) **rollback tetiklemez** -- transaction, exception'a rağmen commit olur:

```
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`, `IllegalStateException` (unchecked) fırlattığı için geri alınıyor. `writeThenThrowChecked`, `IOException` (checked) fırlattığı hâlde **commit oluyor** -- yazılan

satır kalıcı oluyor, exception yalnızca çağırana bildiriliyor. `@Transactional(rollbackFor = IOException.class)` bu varsayılanı açıkça geçersiz kılıp checked bir exception'ı da rollback tetikleyici yapıyor.

> Bu varsayılan kural, tarihsel bir nedenden geliyor (checked exception'lar geleneksel olarak "beklenen, iş akışının bir parçası olan" durumlar, unchecked olanlar "beklenmeyen hatalar" sayılırdı) ama günümüzde birçok ekip bunu kafa karıştırıcı buluyor. Bir checked exception fırlatan `@Transactional` bir metot yazıyorsan, verinin tutarlı kalmasını istiyorsan `rollbackFor` eklemeyi asla unutma -- aksi hâlde exception'ı yakalayan kod, verinin aslında geri alınmadığını fark etmeyebilir.

@EnableTransactionManagement ve Proxy Tabanlı Mekanizma

`@Transactional`, Component Scanning ve Spring IoC Container derslerinde gördüğümüz mekanizmaların hiçbirine benzemiyor -- bir bean tanımlamıyor, bir koşul da belirtmiyor. Bunun yerine, `@EnableTransactionManagement` etkinken, Spring her `@Transactional` içeren bean'in etrafına bir **proxy** sarar:

```
Client
  ↓
Spring Proxy (TransactionInterceptor)
  ↓
transaction başlar
  ↓
Gerçek Metot (Target Method)
  ↓
commit / rollback
```

Bu, Spring AOP'nin (Aspect-Oriented Programming) bir uygulaması -- `TransactionInterceptor`, metot çağrısını gerçek nesneye ulaşmadan önce yakalayan bir "advice." `@Component` / `@Service` gibi component scanning ile bulunan sınıflar için proxy CGLIB (alt sınıf oluşturarak) ya da JDK dynamic proxy (arayüz varsa) ile kurulur -- Spring IoC Container dersinde gördüğümüz `BeanPostProcessor` mekanizmasının (bkz. "Bean Lifecycle: Container'ın Bir Bean'i İnşa Etme Adımları") gerçek bir uygulaması, tam olarak bu şekilde devreye giriyor. Proxy'nin en önemli sonucu bir sonraki bölümün konusu.

Self-Invocation Tuzağı

Proxy, yalnızca **bean üzerinden gelen** çağrıları yakalayabilir -- `this` üzerinden (aynı sınıfın içinden) yapılan bir çağrı hiçbir zaman proxy'den geçmez, bu yüzden `@Transactional` sessizce hiç uygulanmaz:

```
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`, `writeLine(...)`'ı `this.writeLine(...)` olarak çağırıyor -- Spring'in container'dan aldığın proxy nesnesi değil, doğrudan gerçek nesnenin kendisi. `@Transactional` üzerinde dursa da, bu çağrı yolunda hiçbir transaction hiç başlamıyor, dolayısıyla geri alınacak bir şey de yok.

Propagation: REQUIRED (Varsayılan)

`PROPAGATION_REQUIRED`, zaten aktif bir transaction varsa **ona katılır** -- ikinci bir tane başlatmaz. Dış transaction geri alınırsa, çağırdığı her `REQUIRED` metodun yazdığı her şey de onunla birlikte geri alınır, çünkü aslında hepsi baştan beri aynı transaction'dı:

```
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` ve `charge(...)` aynı transaction'ı paylaşıyor -- `charge`'ın kendi `@Transactional`'ı yeni bir transaction başlatmıyor, var olana katılıyor. Ödeme başarıyla "yazılmış" olsa da, sipariş sonradan başarısız olduğunda ikisi birlikte geri alınıyor.

Propagation: REQUIRES_NEW

`PROPAGATION_REQUIRES_NEW`, aktif bir transaction varsa bile onu **askıya alıp** (suspend) tamamen bağımsız, yeni bir transaction başlatır. Bu yeni transaction kendi başına commit ya da rollback olur -- dış transaction daha sonra geri alınsa bile, içteki zaten commit olmuş işi etkilenmez:

```
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` başarısız olup geri alınıyor, ama `recordAuditEntry(...)` -- `REQUIRES_NEW` sayesinde -- kendi ayrı transaction'ında zaten commit olmuştu. Gerçek dünyada bu, tam olarak bir denetim (audit) kaydının neden sıradan bir iş işleminden bağımsız tutulması gerektiğini gösteriyor: "bunu denedik" bilgisi, asıl işlem başarısız olsa bile kalıcı olmalı.

Diğer Propagation Türleri (Kısa Bakış)

Geriye kalan beş propagation türü, bu ortamda çalışan bir örnekle göstermeye değecek kadar sık kullanılmıyor, ama ne yaptıklarını bilmek önemli: **`NESTED`**, dış transaction içinde bir savepoint oluşturur -- iç kısım geri alınabilir, dış kısım etkilenmeden devam edebilir (gerçek bir JDBC savepoint'i gerektirir, bizim `Ledger`'ımız desteklemiyor). **`SUPPORTS`**, aktif bir transaction varsa katılır, yoksa transaction'sız çalışır. **`MANDATORY`**, aktif bir transaction **zorunlu** kılar -- yoksa exception fırlatır. **`NOT_SUPPORTED`**, aktif bir transaction'ı askıya alıp metodu tamamen transaction'sız çalıştırır. **`NEVER`**, aktif bir transaction varsa exception fırlatır -- "bu metot asla bir transaction içinde çağrılmamalı" garantisi.

Isolation Levels (Kısa Bakış)

Isolation, aynı anda çalışan birden fazla transaction'ın birbirinin **henüz commit olmamış** değişikliklerini ne ölçüde görebileceğini belirler -- bu, gerçek eşzamanlı transaction'lar ve gerçek bir veritabanı gerektirdiği için bu derste çalışan bir kod örneği yok, ama çözdüğü üç klasik problemi bilmek önemli: **Dirty Read** (henüz commit olmamış bir değişikliği okumak -- o değişiklik rollback olursa, okuduğun değer hiç var olmamış olur), **Non-Repeatable Read** (aynı satırı aynı transaction içinde iki kez okuyup farklı değerler almak, çünkü aradaki sürede başka bir transaction commit etti), **Phantom Read** (aynı sorguyu iki kez çalıştırıp farklı **satır sayısı** almak, çünkü aradaki sürede başka bir transaction yeni satır ekledi/sildi). Isolation seviyeleri (`READ_UNCOMMITTED`, `READ_COMMITTED`, `REPEATABLE_READ`, `SERIALIZABLE`) bu üç problemi sırasıyla giderek daha sıkı kilitleme/versiyon kontrolü pahasına önler -- `@Transactional(isolation = ...)` ile ayarlanır.

PostgreSQL'de Isolation

Bu projenin veritabanı PostgreSQL, ve PostgreSQL'in isolation davranışının iki gerçekçi özelliği var. Birincisi: PostgreSQL'in **varsayılan** isolation seviyesi `READ_COMMITTED`'dır (Spring /JPA'nın kendi varsayılanı `Isolation.DEFAULT` de zaten bunu miras alır -- yani bu proje, hiçbir şey özelleştirmeden, zaten `READ_COMMITTED` ile çalışıyor). İkincisi, ve daha az bilinen: PostgreSQL, `READ_UNCOMMITTED`'ı **gerçekten desteklemez** -- `READ_UNCOMMITTED` istesen bile, motor sessizce `READ_COMMITTED`'a yükseltir; yani PostgreSQL'de "dirty read" hiçbir zaman gerçekleşmez, isteğe bağlı bile olsa. `REPEATABLE_READ` ve `SERIALIZABLE`, PostgreSQL'de

kilitleme yerine "snapshot isolation" (MVCC) ile uygulanır -- `SERIALIZABLE`'da bir çakışma tespit edilirse, transaction commit anında serialization hatasıyla başarısız olabilir; uygulama kodunun bu durumda yeniden denemesi (retry) gerekir.

readOnly = true: Ne İşe Yarar, Ne İşe Yaramaz

`@Transactional(readOnly = true)`, Spring'e ve alttaki JPA/Hibernate'e bir **ipucu** verir -- gerçek bir kısıtlama değildir:

```
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`, `readOnly = true` olmasına rağmen sorunsuz yazıyor -- ne Spring'in `@Transactional` sözleşmesi ne de bizim basit `LedgerTransactionManager`'ımız bunu engelliyor. Gerçek `JpaTransactionManager`'da `readOnly = true`'nun asıl faydası performans: Hibernate'in "dirty checking" (bkz. "Spring Data JPA ve Dirty Checking") mekanizmasını devre dışı bırakarak flush'ı atlar, ve bazı JDBC sürücüleri bunu okumaları bir replica'ya yönlendirmek için kullanır. Ama "bu metot kesinlikle veritabanına yazamaz" **anlamına gelmez** -- bu güvenceyi istiyorsan, veritabanı kullanıcısının kendisine salt okunur yetki vermen gerekir.

Transaction Boundary: Neden Service Katmanında?

Bir transaction nerede başlamalı? Tipik bir katmanlı mimaride:

```
Controller
    ↓
Service   ← transaction boundary burada
    ↓
Repository
```

`@Transactional`'ı **service** katmanına koymak genel kabul görmüş kuraldır, iki sebeple: birincisi, tek bir service metodu genellikle birden fazla repository çağrısı yapar (bkz. "Propagation: REQUIRED (Varsayılan)" örneğindeki `OrderService` -> `PaymentService`) -- transaction boundary'yi burada çizmek, bu çağrıların hepsinin tek bir birim olmasını sağlar. İkincisi, controller katmanına `@Transactional` koymak (bkz. "Yaygın Hatalar") transaction'ı

gereğinden geniş tutar -- view render etme, JSON serialize etme gibi veritabanıyla ilgisi olmayan işler de transaction içinde kalır.

Programmatic Transactions: TransactionTemplate

`TransactionTemplate`, `@Transactional`'ın programatik karşılığı -- transaction sınırının "metodun tamamı" olmadığı, ya da koşullu olması gereken durumlar için:

```
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`, lambda'nın tamamını bir transaction içinde çalıştırıyor -- normal dönerse commit, exception fırlatırsa rollback, `@Transactional` ile aynı kural. `status.setRollbackOnly()` ise farklı bir yol sunuyor: hiçbir exception fırlatmadan, sadece bir iş kuralı gereği transaction'ı geri almak istediğinde kullanılır.

Spring Data JPA ve Dirty Checking

Spring Data JPA'nın kendi repository metotları (`save()`, `findById()`, `delete()` gibi -- Component Scanning dersindeki "Bu Projenin Kendi Sınıfları: Gerçek Bir Component Scanning Örneği" bölümünde gördüğümüz, hiç `@Repository` yazmadan proxy olarak üretilen `TopicRepository` gibi arayüzler) zaten kendi içlerinde `@Transactional`'dır (`SimpleJpaRepository` üzerinde tanımlı). Bunun ötesinde, Hibernate'in **dirty checking** özelliği, bir transaction içinde yönetilen (managed) bir entity'nin alanları değiştirildiğinde, `save()`'i hiç çağırmadan bile bu değişikliğin commit'te veritabanına yazılmasını sağlar. Bu proje tamamen salt-okunur olduğu için (bkz. "Bu Projenin Kendi Repository'leri: Neden Hâlâ @Transactional Yok?") gerçek bir örneği yok, ama varsayımsal olarak: eğer bir `TopicService.updateDifficulty(slug, yeniZorluk)` metodu olsaydı ve içinde `topicRepository.findBySlug(slug)` ile alınan yönetilen bir `Topic` üzerinde `topic.setDifficulty(yeniZorluk)` çağırsaydık, `topicRepository.save(topic)`'i **hiç çağırmasak bile**, transaction commit olduğunda Hibernate bu değişikliği fark edip bir `UPDATE` sorgusu gönderirdi.

> Dirty checking, `readOnly = true` transaction'larda devre dışı bırakılır (bkz. "readOnly = true: Ne İşe Yarar, Ne İşe Yaramaz") -- Hibernate, hiçbir zaman yazılmayacağını bildiği entity'ler için flush öncesi karşılaştırmayı atlayarak performans kazanır.

Lazy Loading ve LazyInitializationException

Bu projenin `Topic`, `Category`, `TopicTranslation`, `CodeExample` entity'lerinin hepsinde gerçek `@ManyToOne(fetch = FetchType.LAZY)` ilişkileri var (`TopicTranslation.topic`, `CodeExample.topic`, `Category.course`, `Topic.category`). Lazy bir ilişki, yalnızca **açıkça erişildiğinde** (örn. `topic.getCategory()`) veritabanından çekilir -- ve bu erişim, entity'nin bağlı olduğu persistence context (Hibernate session) hâlâ açıkken olmalıdır. Kapandıktan sonra erişmeye çalışmak `LazyInitializationException` fırlatır.

`TopicRepository`'nin gerçek kaynak kodunda tam olarak bu problemden kaçınan bir metot var:

```
@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(...)`, sıradan `findBySlug(slug)` yerine bilerek bunu kullanıyor -- `join fetch`, `category` ve `course` ilişkilerini **aynı sorguda**, tembel yükleme beklemeden hemen getiriyor. Kaynak koddaki yorum bunu açıkça söylüyor: breadcrumb ve önceki/sonraki konu navigasyonu bu ilişkilere ihtiyaç duyuyor, ve proje bunu "lazy-loading'e (open-in-view'a) bırakmak yerine tek sorguda açıkça çözüyor." `spring.jpa.open-in-view` bu projede hiç ayarlanmadığı için Spring Boot'un varsayılanı (`true`) geçerli -- yani `findBySlug(slug)` kullansaydı ve Thymeleaf şablonu `topic.category.course.name`'e erişseydi muhtemelen yine de çalışırdı (open-in-view, persistence context'i view render'ı bitene kadar açık tutar), ama bu, veritabanı bağlantısını gereğinden uzun süre elde tutan, genel kabul görmüş bir anti-pattern'dir -- projenin `join fetch` tercihi tam olarak bunu önlüyor.

Transactional Events: @TransactionalEventListener ve AFTER_COMMIT

Auto-Configuration & Properties dersinde `ApplicationEvent` / `@EventListener`'ı gördük -- `@TransactionalEventListener`, aynı fikri transaction'a duyarlı hâle getiriyor: bir olayı, yayınlandığı anda değil, transaction belirli bir aşamaya ulaştığında işler. En sık kullanılan aşama `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`, `simulateFailureAfterPublish = false` olduğunda başarıyla commit oluyor ve dinleyici çalışıyor. `true` olduğunda ise, event yayınlanmış olsa bile, transaction hiç commit olmadığı için `AFTER_COMMIT` dinleyicisi **hiç çalışmıyor** -- olay yayınlamak ile o olayın etkisinin gerçekleşmesi arasındaki fark burada net görülüyor.

Testing Transactions (Kısa Bakış)

Spring Test (`spring-boot-starter-test` , bu projenin bir bağımlılığı), bir test metoduna `@Transactional` koyduğunda özel bir davranış ekler: test metodu kendi transaction'ında çalışır ve **test bittiğinde otomatik olarak rollback olur** -- varsayılan olarak, testin veritabanına yazdığı hiçbir şey kalıcı olmaz, bir sonraki test temiz bir veritabanıyla başlar:

```
@SpringBootTest
class OrderServiceTest {

    @Test
    @Transactional
    void shouldCreateOrder() {
        // ... veritabanına yazan kod ...
        // test bittiğinde otomatik rollback, elle temizlik gerekmez
    }
}
```

Bu, `TransactionalTestExecutionListener` tarafından sağlanır ve gerçek bir Spring Boot test ortamı (`@SpringBootTest`) gerektirir -- bu dersteki diğer örneklerin kullandığı sade `AnnotationConfigApplicationContext` + `main()` biçiminden farklı bir çalıştırma modeli olduğu için burada ayrı bir kod örneği yok. Dikkat edilmesi gereken bir nokta: eğer test edilen kodun kendisi `REQUIRES_NEW` kullanıyorsa (bkz. "Propagation: REQUIRES_NEW"), o iç transaction test'in kendi transaction'ından bağımsız olarak **gerçekten commit olur** -- test'in dış rollback'i bunu geri alamaz.

Bu Projenin Kendi Repository'leri: Neden Hâlâ @Transactional Yok?

Bu proje boyunca hiçbir yerde `@Transactional` yok -- `grep -rn "@Transactional" src/main/java` boş dönüyor. Sebebi basit: `NavigationService` , `ContentResolver` , `TopicController` gibi bu projenin gerçek sınıflarının hepsi **salt okuma** yapıyor -- `courseRepository.findAll()` , `topicRepository.findBySlugWithCategoryAndCourse(...)` gibi tekil, tek-sorguluk repository çağrıları. Spring Data JPA'nın kendi `SimpleJpaRepository` 'si (bkz. "Spring Data JPA ve Dirty Checking") her repository metodunu zaten kendi başına bir transaction'a sarıyor -- tek bir sorgu için ayrıca servis katmanına `@Transactional` eklemenin hiçbir faydası olmazdı.

Projenin gördüğü tek "yazma" işlemleri, çalışan uygulamanın kendisinden değil, Flyway migration'larından geliyor (bkz. `db/migration/`) -- her `INSERT` / `UPDATE` , uygulama her başladığında, servis katmanı hiç devreye girmeden, doğrudan SQL olarak çalışıyor. Eğer bu projeye gelecekte bir "admin panel" ya da içerik düzenleme özelliği eklenseydi (örneğin bir konunun `sort_order` 'ını değiştiren bir `TopicService.reorder(...)` metodu), işte tam o zaman gerçek bir `@Transactional` service metoduna ihtiyaç duyulurdu -- muhtemelen birden fazla `Topic` satırını tek bir birim olarak güncelleyen, "Propagation: REQUIRED (Varsayılan)" bölümündeki `OrderService` örneğine çok benzer bir yapıda.

Best Practices

- **`@Transactional` 'ı service katmanına koy, controller'a değil** -- transaction sınırının yalnızca veritabanıyla ilgili işleri kapsamasını sağlar (bkz. "Transaction Boundary: Neden Service Katmanında?").
- **Checked exception fırlatan bir `@Transactional` metotta `rollbackFor` 'u unutma** -- varsayılan davranış, checked exception'larda commit yapar, bu genellikle istenen şey değildir (bkz. "Rollback Kuralları: RuntimeException vs Checked Exception").
- **Salt okuma yapan metotları `readOnly = true` işaretle** -- gerçek bir kısıtlama sağlamasa da, gerçek `JpaTransactionManager` 'da performans kazandırır (bkz. "readOnly = true: Ne İşe Yarar, Ne İşe Yaramaz").
- **Transaction'ları kısa tut, içine harici bir API çağrısı koyma** -- bir ödeme sağlayıcısı ya da e-posta servisi gibi harici bir çağrı yavaş/başarısız olursa, veritabanı bağlantısını

(ve varsa kilitleri) gereğinden uzun süre açık tutar; böyle işler için `@TransactionalEventListener(phase = AFTER_COMMIT)` daha uygundur (bkz. "Transactional Events: @TransactionalEventListener ve AFTER_COMMIT").

- **Gereksiz yere `REQUIRES_NEW` kullanma** -- her `REQUIRES_NEW` çağrısı ayrı bir transaction (ve gerçek bir veritabanında ayrı bir bağlantı) demektir; yalnızca denetim kaydı gibi, dış transaction'dan gerçekten bağımsız olması gereken işler için kullan (bkz. "Propagation: REQUIRES_NEW").

Yaygın Hatalar

1. `@Transactional`'ı controller'a koymak. Transaction sınırı, view render etme gibi veritabanıyla ilgisi olmayan işleri de kapsar hâle gelir -- doğrusu service katmanıdır (bkz. "Transaction Boundary: Neden Service Katmanında?").

2. Self-invocation ile `@Transactional`'ın çalışacağını sanmak. `this` üzerinden yapılan bir çağrı proxy'den hiç geçmez -- annotation sessizce hiçbir işe yaramaz (bkz. "Self-Invocation Tuzağı").

3. Checked bir exception fırlatan bir metodun otomatik rollback yapacağını varsaymak. Varsayılan davranış tam tersi: checked exception'lar commit'e izin verir, `rollbackFor` açıkça yazılmadıkça (bkz. "Rollback Kuralları: RuntimeException vs Checked Exception").

4. `readOnly = true`'nun "bu metot yazamaz" garantisi verdiğini sanmak. Bu yalnızca bir performans ipucudur, bir kısıtlama değildir (bkz. "readOnly = true: Ne İşe Yarar, Ne İşe Yaramaz").

5. Isolation seviyesini ne yaptığını tam anlamadan değiştirmek. Daha sıkı bir seviye (örn. `SERIALIZABLE`) eşzamanlılık problemlerini önler ama performansı düşürür ve serialization hatalarına (özellikle PostgreSQL'de) yol açabilir -- varsayılanı değiştirmeden önce hangi problemi (dirty read, non-repeatable read, phantom read) çözmeye çalıştığını netleştir (bkz. "Isolation Levels (Kısa Bakış)").

6. Lazy loading problemini her yere `@Transactional`/open-in-view yayarak "çözmeye" çalışmak. Bu, veritabanı bağlantısını gereğinden uzun süre açık tutar -- doğru çözüm, bu projenin `findBySlugWithCategoryAndCourse`'unda olduğu gibi, ihtiyaç duyulan ilişkileri `join fetch` ile açıkça, tek sorguda getirmektir (bkz. "Lazy Loading ve LazyInitializationException").

Özet, Cheat Sheet ve Terimler Sözlüğü

Transaction management, birden fazla veritabanı işlemini tek, bölünemez bir birim hâline getirir -- `@Transactional`, bunu bir proxy aracılığıyla, metot çağrısını sarıp exception fırlatılıp fırlatılmadığına göre commit/rollback kararı vererek sağlar. Önemli noktalar:

- ACID: Atomicity, Consistency, Isolation, Durability

- Varsayılan rollback kuralı: unchecked exception (`RuntimeException` / `Error`) rollback tetikler, checked exception tetiklemez -- `rollbackFor` ile değiştirilir
- `@Transactional`, bir proxy (AOP, `TransactionInterceptor`) aracılığıyla çalışır -- self-invocation (`this` üzerinden çağrı) bu proxy'yi atlar
- Propagation: `REQUIRED` (varsayılan, var olana katılır), `REQUIRES_NEW` (askıya alıp bağımsız yeni bir tane başlatır), `NESTED` / `SUPPORTS` / `MANDATORY` / `NOT_SUPPORTED` / `NEVER` (daha az kullanılan diğer türler)
- Isolation levels, dirty read/non-repeatable read/phantom read problemlerini giderek daha sıkı biçimde önler; PostgreSQL varsayılanı `READ_COMMITTED`
- `readOnly = true`: performans ipucu, kısıtlama değil
- `TransactionTemplate`: `@Transactional`'ın programatik karşılığı
- `@TransactionalEventListener(phase = AFTER_COMMIT)`: bir event'i, yalnızca yayınlandığı transaction gerçekten commit olursa işler

Hızlı referans:

```
@Transactional                                      // REQUIRED, tüm exception rollback
(varsayılan hariç checked)
@Transactional(rollbackFor = Exception.class)      // checked exception'ları da rollback
tetikleyici yap
@Transactional(readOnly = true)                    // performans ipucu, kısıtlama değil
@Transactional(propagation = Propagation.REQUIRES_NEW)  // her zaman yeni, bağımsız
transaction
@Transactional(isolation = Isolation.SERIALIZABLE) // en sıkı izolasyon

class MyService {
    // Self-invocation UYARISI: this.otherMethod() proxy'yi atlar.
    void outer() {
        this.inner(); // @Transactional'ı olsa bile UYGULANMAZ
    }

    @Transactional
    void inner() { }
}

// Programatik alternatif:
transactionTemplate.executeWithoutResult(status -> {
    // ...
    if (someCondition) {
        status.setRollbackOnly(); // exception fırlatmadan rollback
    }
});

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

Terimler Sözlüğü

Transaction — Birden fazla işlemin ya hep birlikte (commit) ya da hiç (rollback) gerçekleşmesini garanti eden, bölünemez bir birim.

ACID — Atomicity, Consistency, Isolation, Durability: bir transaction'ın sağlaması gereken dört özellik.

`@Transactional` — Bir metodu (ya da sınıfı) bir transaction sınırıyla saran annotation.

`PlatformTransactionManager` — Spring'in transaction başlatma/commit/rollback işlemlerini soyutlayan arayüzü; `DataSourceTransactionManager`, `JpaTransactionManager` gibi gerçek implementasyonları vardır.

Rollback kuralı — Hangi exception türlerinin rollback tetikleyeceğini belirleyen kural; varsayılan olarak yalnızca unchecked exception'lar.

Self-invocation — Bir proxy'lenmiş bean'in kendi metodunu `this` üzerinden çağırması; proxy'yi atladığı için `@Transactional` gibi proxy'ye dayalı annotation'ları devre dışı bırakır.

Propagation — Bir `@Transactional` metodun, zaten aktif bir transaction varken nasıl davranacağını belirleyen ayar (`REQUIRED`, `REQUIRES_NEW`, vb.).

Isolation — Aynı anda çalışan transaction'ların birbirinin commit olmamış değişikliklerini ne ölçüde görebileceğini belirleyen ayar.

`readOnly` — Bir transaction'ın yalnızca okuma yapacağını belirten, performans amaçlı bir ipucu; bir kısıtlama değildir.

`TransactionTemplate` — `@Transactional`'ın programatik (annotation'sız) karşılığı.

`@TransactionalEventListener` — Bir event'i, yayınlandığı transaction belirli bir aşamaya (en sık `AFTER_COMMIT`) ulaştığında işleyen dinleyici annotation'ı.

Ek: Mini Proje — Para Transferi

Bu mini proje, dersin baştan beri kullandığı hesap transferi senaryosunu, rollback kurallarını, `PROPAGATION_REQUIRED`'ı ve self-invocation tuzağını bir araya getirerek tamamlıyor:

```
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(...)`, `debit(...)` ve `credit(...)`'in `PROPAGATION_REQUIRED` sayesinde aynı transaction'ı paylaşmasına dayanıyor -- bakiyesi yetersiz bir hesaptan çekim denendiğinde, `debit(...)` kendi `InsufficientFundsException`'ını (unchecked) fırlatıyor ve her şey geri alınıyor. `transferViaSelfInvocation(...)` ise bilerek bozuk: `transferInternal(...)`'ı `this` üzerinden çağırdığı için, o metodun `@Transactional`'ı hiç uygulanmıyor -- ama `debit(...)` / `credit(...)`'in kendi `@Transactional`'ları (ayrı bir bean olan `accountRepository` üzerinden çağrıldıkları için) normal şekilde işliyor ve her biri kendi başına commit oluyor.

Ek: Mini Proje — Sipariş İşleme

Son mini proje, gerçekçi bir `OrderService` -> `PaymentService` -> `InventoryAuditService` akışında propagation'ı ve transactional event'leri bir araya getiriyor:

```
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();
    }
}
```

Sipariş ve ödeme aynı transaction'ı (`REQUIRED`) paylaşırken, denetim kaydı (`recordAttempt`) bilerek `REQUIRES_NEW` -- sipariş daha sonra stok yetersizliği yüzünden geri alınsa bile, "bu siparişi denedik" bilgisi kalıcı kalıyor. Kargo bildirimi ise yalnızca `AFTER_COMMIT` 'te tetikleniyor -- stok yetersizliği durumunda hiçbir bildirim gönderilmiyor, çünkü o transaction hiç commit olmuyor.

> `InventoryAuditService.recordAttempt(...)` 'in her çağrısı gerçek bir veritabanında ayrı bir bağlantı/transaction açar -- "Best Practices" bölümünde belirtildiği gibi, `REQUIRES_NEW` 'i yalnızca gerçekten bağımsız olması gereken işler için kullanmak, performans ve bağlantı havuzu tükenmesi açısından önemlidir.