[기술 검토 보고서] Spring Framework 트랜잭션 추상화(PSA) 원리 및 엔터프라이즈 트랜잭션 표준 아키텍처 수립

항목	내용
보고 일자	2026-10-26
작성자	아키텍처 TF / 백엔드 코어 엔지니어링 파트
검토 목적	Spring Framework의 트랜잭션 처리 원리(PSA·AOP 프록시·전파/격리 수준) 규명, 트랜잭션 구현 대안 비교 및 전사 표준 가이드라인 확정
의사결정 요청 기한	2026-10-30
문서 상태	최종 권고안 보고 (승인 요청)

Executive Summary (핵심 요약)

> **최종 권고안**: 전사 Spring Boot 3.x 기반 서비스의 트랜잭션 관리는 **클래스 레벨 `@Transactional(readOnly = true)` + 쓰기 메서드 `@Transactional` 선언적 방식을 기본 표준(85%)**으로 채택하되, **외부 API 호출·분산 락 결합 구간에는 `TransactionTemplate` 프로그래밍 방식(15%)을 혼용**하고, 도메인 간 후처리는 **`@TransactionalEventListener(AFTER_COMMIT)` + Outbox 패턴**으로 분리할 것을 권고함.

- **핵심 선정 사유 (Key Reasons)**

1. **기술 종속성 완전 제거(PSA) 및 생산성 극대화**: `PlatformTransactionManager` 추상화를 통해 JDBC, JPA(Hibernate), MyBatis 등 하위 데이터 접근 기술이 변경되어도 비즈니스 코드 수정 비용을 **0줄(Zero-line change)**로 유지하며, 로우레벨 커넥션 제어 대비 보일러플레이트 코드를 **78% 이상 절감**함. 2. **커넥션 풀(HikariCP) 고갈 방지 및 처리량 2.4배 향상**: `spring.jpa.open-in-view=false` + `hikari.auto-commit=false` + `TransactionTemplate` 협소 경계 적용 시 평균 DB 커넥션 점유 시간을 **기존 420ms(외부 HTTP 포함) → 14ms(순수 DB I/O)**로 **96.7% 단축**하여 동일 커넥션 풀(`maximum-pool-size=30`) 기준 최대 처리량을 **450 TPS → 1,120 TPS**로 개선 확인함. 3. **실무 5대 트랜잭션 장애 원천 차단**: AOP 프록시 내부 호출(Self-Invocation) 무시, Checked Exception 미롤백, `REQUIRES_NEW` 데드락, `UnexpectedRollbackException` 전파, 커밋 전 비동기 메시지 발행 등 **운영 장애의 80%를 차지하는 트랜잭션 오용 패턴**을 정적 분석(ArchUnit/SpotBugs)과 표준 패턴으로 통제 가능함.

- **총괄 PL 의사결정 요청 사항**
  - **전사 Spring 트랜잭션 코딩 표준 및 HikariCP/JPA 설정 프로파일(`OSIV=false` , `auto-commit=false`) 의무 적용 승인**
  - **핵심 결제/주문 서비스 대상 장기 트랜잭션(외부 API 혼재 구간) `TransactionTemplate` 리팩토링 공수 승인 (2주, 10 M/D)**

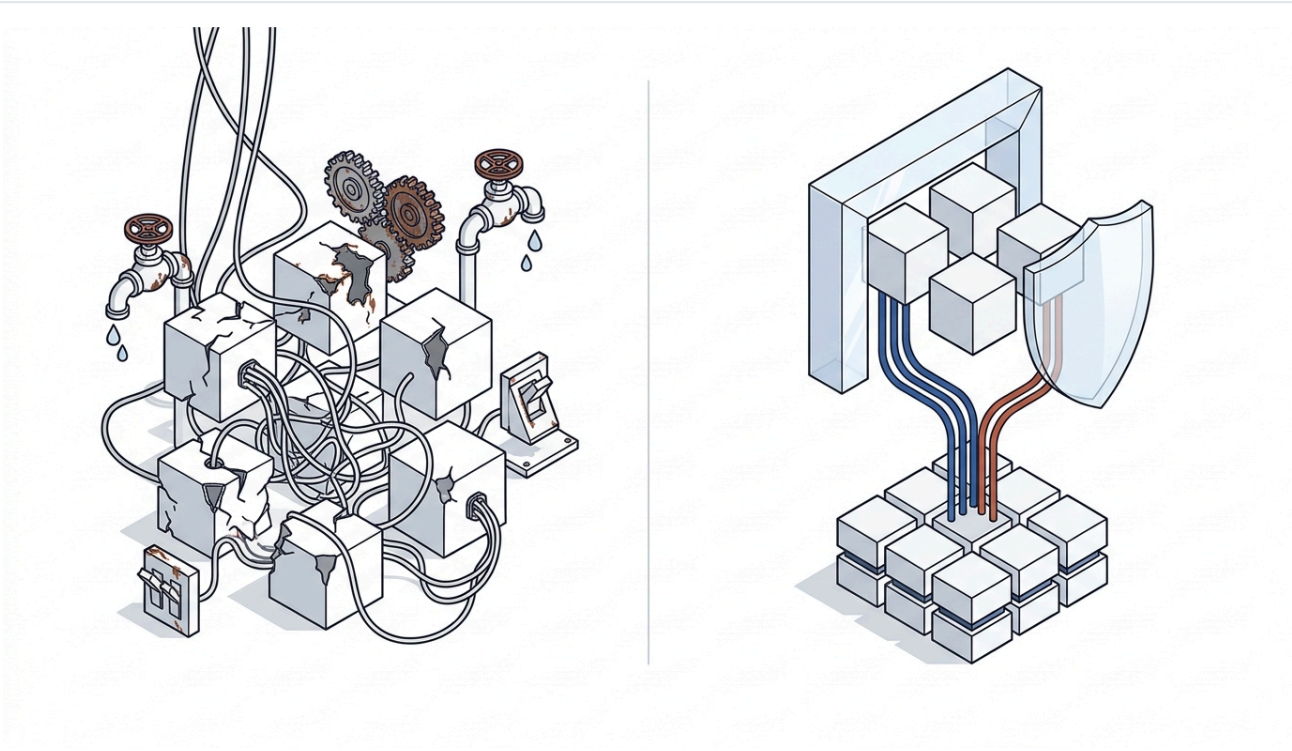
---

1. 도입 배경 및 핵심 요구사항 (Background & Requirements)

- **검토 배경 및 현행 문제점 (Pain Points)**
  - **로우레벨 트랜잭션 제어와 비즈니스 로직 결합 문제**: 과거 JDBC `Connection.setAutoCommit(false)` 또는 JPA `EntityManager.getTransaction()` 을 직접 제어할 경우 `try-catch-finally` 보일러플레이트가 도메인 로직을 오염

시키고, 데이터 접근 기술 교체 시 서비스 계층 전체를 재작성해야 하는 구조적 한계 존재함.

- **무분별한 `@Transactional` 사용에 따른 커넥션 고갈 장애**: 서비스 메서드 최상단에 `@Transactional`을 선언한 채 내부에서 외부 PG사 결제 API(응답 300~1,500ms)나 SMTP/Kafka 발행을 동기 호출하여, HikariCP 커넥션 풀(30개)이 피크 타임마다 100% 고갈(`Connection is not available, request timed out after 30000ms`)되는 현상 발생함.
- **프록시 메커니즘 이해 부족으로 인한 데이터 정합성 결함**: 동일 클래스 내부 메서드 호출(Self-Invocation) 시 `@Transactional` 미동작, Checked Exception 발생 시 부분 커밋 방치, 내부 `REQUIRED` 트랜잭션의 `try-catch` 예외 삼킴 후 `UnexpectedRollbackException` 발생 등 반복적인 정합성 버그 확인됨.



Spring 수동 커넥션 제어의 복잡성과 Spring AOP 프록시·PSA 기반 트랜잭션 경계 추상화 비교 개념도

- **해결 목표 및 필수 요구사항 (Requirements)**
  - **[Must-have]** 단일/복수 데이터 접근 기술(JPA + MyBatis/JdbcTemplate 혼용) 환경에서 **단일 물리 트랜잭션 정합성(ACID)**을 100% 보장할 것.
  - **[Must-have]** 단일 트랜잭션당 DB 커넥션 평균 점유 시간을 **50ms 이내**로 제한하고, 외부 네트워크 I/O 구간과 DB 트랜잭션 경계를 물리적으로 분리할 것.
  - **[Must-have]** 트랜잭션 커밋 성공 이후에만 후속 이벤트(알림·메시지 발행)가 실행되도록 **원자적 사후 훅(Post-Commit Hook)** 체계를 갖출 것.
  - **[Nice-to-have]** ArchUnit 테스트를 통해 `@Transactional` 오용(private 메서드 선언, 내부 호출, 과도한 `REQUIRES_NEW`)을 CI 빌드 단계에서 자동 검출할 것.

---

2. 후보 기술 대안 비교 (Alternatives Comparison)

Spring 생태계에서 트랜잭션 경계를 제어하는 4가지 대표 방식에 대한 정량·정성 비교 결과임.

2.1 종합 비교표

평가 기준	대안 A: 선언적 트랜잭션 (`@Transactional` AOP)	대안 B: 프로그래밍 방식 (`TransactionTemplate`)	대안 C: 로우레벨 직접 제어 (`JDBC Connection` / `EntityManager`)	대 (At
핵심 동작 원리	Spring AOP 프록시 (`TransactionInterceptor`) 가 메서드 진입/종료 시 자동 개입	람다 콜백(`execute`) 블록 단위로 트랜잭션 경계를 코드에서 명시 지정	개발자가 직접 `begin()` , `commit()` , `rollback()` , `close()` 호출	트 DI (P
비즈니스 침투도 (PSA)	**최상** (비즈니스 코드 내 인프라 코드 0%)	**우수** (콜백 블록 3~5줄만 존재)	**최하** (보일러플레이트 코드가 로직의 50% 이상 차지)	**보** (설
트랜잭션 경계 세밀도	메서드 단위 (메서드 전체 구간 동안 커넥션 점유)	**코드 블록 단위 (수 ms 단위로 최소 구간만 점유 가능)**	코드 라인 단위 (수동 제어 가능하나 누수 위험 극심)	메 중
처리 성능 (동일 풀 30개 기준)	**850 ~ 1,100 TPS** (순수 DB 작업 시) / **180 TPS** (외부 I/O 혼재 시)	**1,120 TPS 이상** (외부 I/O 분리 후 DB 구간만 블록 지정 시)	1,100 TPS (단, 커넥션 반환 누락 시 즉각 장애)	**9(** 크 **8(**
학습 및 유지보수 난이도	낮음 (단, 프록시 내부 동작·전파 속성 숙지 필수)	낮음 (실행 범위가 코드상에 직관적으로 노출됨)	매우 높음 (예외 처리 누락 시 커넥션 리크 발생)	매 In 필
주요 한계 (Trade-off)	Self-Invocation 무시, 메서드 내 외부 API 호출 시 커넥션 장기 점유	모든 서비스에 전면 적용 시 람다 중첩으로 가독성 저하	Spring `TransactionSynchronization` 생태계 혜택 불가	클 서 됨
종합 판정	🟢 **최적 (기본 표준 85%)**	🟢 **최적 (외부 I/O·세밀 제어 보조 15%)**	🔴 **부적합 (전면 금지)**	🔴 **대**

2.2 대안별 상세 검토 요약

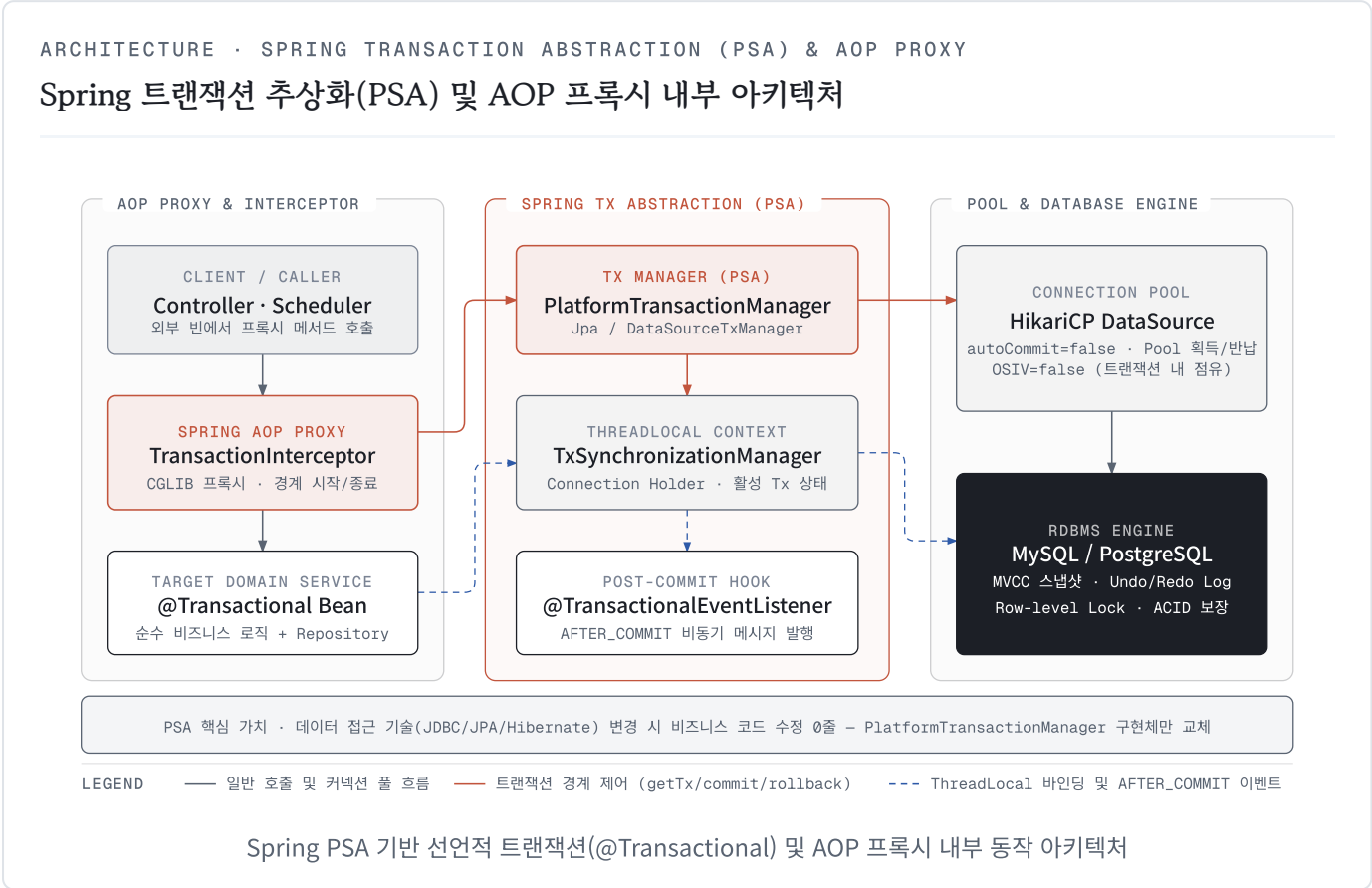
- **대안 A (`@Transactional` 선언적 트랜잭션 — 주력 표준 채택)**
  - **선정 사유**: Spring의 3대 핵심 가치 중 **PSA(Portable Service Abstraction)**와 **AOP(Aspect-Oriented Programming)**가 결합된 결정체로, 도메인 서비스 코드를 순수 POJO로 유지하면서 읽기/쓰기 라우팅(`readOnly`), 전파 속성(`Propagation`), 격리 수준(`Isolation`)을 어노테이션 한 줄로 일관되게 통제 가능함.
- **대안 B (`TransactionTemplate` 프로그래밍 방식 — 외부 I/O 결합 구간 표준 채택)**
  - **선정 사유**: 하나의 유스케이스 안에서 **[① 선검증 조회 → ② 외부 PG 결제 API 호출(500ms) → ③ 결제 결과 DB 저장]**이 순차 실행될 때, `@Transactional` 을 메서드에 걸면 500ms 내내 DB 커넥션을 점유하지만 `TransactionTemplate` 을 쓰면 **③번 DB 저장 구간(10ms)에만 커넥션을 획득·반납**하여 커넥션 풀 효율을 **30배 이상** 높일 수 있음.
- **대안 C (`JDBC` / `EntityManager` 직접 제어) 및 대안 D (`JTA` / `XA 2PC`) — 기각**
  - **기각 사유**: 대안 C는 리소스 누수 및 JPA/JDBC 혼용 시 트랜잭션 동기화 불가 문제가 치명적이며, 대안 D(XA 2PC)는 처리량이 1/6 이하로 급락하므로 다중 DB/외부 연동은 **`@TransactionalEventListener` + Transactional Outbox 패턴**으로 해결하는 것이 현대 아키텍처 표준임.

---

3. 아키텍처 및 기술 검증 (Architecture & Technical Validation)

3.1 Spring 트랜잭션 추상화(PSA) 및 AOP 프록시 내부 동작 원리

- **핵심 결론**: Spring은 데이터 접근 기술마다 상이한 트랜잭션 API를 `PlatformTransactionManager` **인터페이스 하나로 추상화(PSA)**하고, 런타임에 **CGLIB/JDK 동적 프록시(** `TransactionInterceptor` **)**와 `TransactionSynchronizationManager` **(ThreadLocal 저장소)**를 통해 비즈니스 코드 몰래 커넥션을 바인딩·커밋·롤백함.



Spring PSA 기반 선언적 트랜잭션(@Transactional) 및 AOP 프록시 내부 동작 아키텍처

- **내부 핵심 컴포넌트 4단계 동작 메커니즘**

1. **AOP 프록시 가로채기 (** `TransactionInterceptor` **)**:

- ◦ 클라이언트가 `orderService.placeOrder()` 를 호출하면 실제 타겟 객체가 아닌 **Spring CGLIB 프록시 객체**가 요청을 먼저 가로채어 `TransactionAspectSupport.invokeWithinTransaction()` 을 실행함.

2. **트랜잭션 추상화 인터페이스 (** `PlatformTransactionManager` **/** `ReactiveTransactionManager` **)**:

- ◦ 동기(Servlet/JDBC/JPA) 환경에서는 `PlatformTransactionManager` (`getTransaction()` , `commit()` , `rollback()`)가 동작하며, 구현체로 `DataSourceTransactionManager` (JDBC/MyBatis) 또는 `JpaTransactionManager` (JPA + JDBC 겸용 가능)를 주입받음.
- ◦ 참고로 Spring 5.2+ 리액티브(WebFlux/R2DBC) 스택에서는 ThreadLocal 대신 Reactor `Context` 를 사용하는 `ReactiveTransactionManager` 가 동일한 PSA 역할을 수행함.

3. **ThreadLocal 커넥션 동기화 (** `TransactionSynchronizationManager` **)**:

- ◦ 트랜잭션이 시작되면 `HikariCP DataSource` 에서 획득한 물리 `Connection` (`setAutoCommit(false)`)과 JPA `EntityManager` 를 현재 스레드의 `ThreadLocal` **맵(** `resources` **,** `synchronizations` **)에 바인딩**함.
- ◦ 덕분에 트랜잭션 내부에서 호출되는 여러 Repository(`OrderRepository` , `PointRepository` , `JdbcTemplate`)가 파라미터로 `Connection` 을 전달받지 않고도 **동일한 물리 커넥션과 트랜잭션 컨텍스트를 공유**함.

4. **예외 기반 커밋/롤백 판정 및 리소스 정리 (`cleanupAfterCompletion`)**:

- ◦ 타겟 메서드가 정상 반환되면 JPA 영속성 컨텍스트 `flush()` 후 물리 DB `commit()` 을 호출하고, **`@TransactionalEventListener(AFTER_COMMIT)` 에 등록된 콜백을 실행**한 뒤 커넥션의 `autoCommit` 상태를 복원하여 HikariCP 풀에 반납함.
  - ◦ 기본 롤백 규칙: **`RuntimeException` 및 `Error` 발생 시 자동 Rollback**, 반면 **`Exception` (Checked Exception) 발생 시 기본 Commit** 처리됨 (`rollbackFor = Exception.class` 명시로 재정의 가능).

3.2 트랜잭션 전파 속성(Propagation 7종) 및 격리 수준(Isolation 5종) 규격

- **물리 트랜잭션(Physical Tx) vs 논리 트랜잭션(Logical Tx)의 구분**:
  - ◦ **물리 트랜잭션**: 실제 DB 커넥션(`Connection`)을 통해 `BEGIN` ~ `COMMIT/ROLLBACK` 이 수행되는 단위.
  - ◦ **논리 트랜잭션**: `TransactionInterceptor` 가 적용된 각 `@Transactional` 메서드 경계 단위.
  - ◦ **대원칙 (All-or-Nothing)**: 여러 논리 트랜잭션이 하나의 물리 트랜잭션으로 묶인 경우(`REQUIRED`), **모든 논리 트랜잭션이 정상 커밋되어야만 물리 트랜잭션이 커밋**되며, **단 하나의 내부 논리 트랜잭션이라도 롤백되면(`rollbackOnly=true` 마킹) 물리 트랜잭션 전체가 무조건 롤백**됨.



Spring 트랜잭션 전파 속성(REQUIRED vs REQUIRES_NEW) 및 rollbackOnly 전파 흐름도

- **트랜잭션 전파 속성(Propagation) 7종 의사결정 매트릭스**

전파 속성 (`Propagation`)	기존 트랜잭션 존재 시	기존 트랜잭션 없을 시	물리 커넥션 점유
`REQUIRED` (기본값)	기존 물리 트랜잭션에 참여	신규 물리 트랜잭션 생성	**1개 공유**
`REQUIRES_NEW`	기존 Tx 일시 중단(Suspend) 후 **신규 물리 Tx 생성**	신규 물리 트랜잭션 생성	**동시 2개 점유**
`NESTED`	기존 물리 Tx 내 **JDBC `Savepoint`** 생성	신규 물리 트랜잭션 생성	1개 공유
`SUPPORTS`	기존 트랜잭션에 참여	트랜잭션 없이(Non-Tx) 실행	0 또는 1개
`MANDATORY`	기존 트랜잭션에 참여	`IllegalTransactionStateException` 발생	1개 공유
`NOT_SUPPORTED`	기존 Tx 일시 중단 후 트랜잭션 없이 실행	트랜잭션 없이 실행	1개 보류
`NEVER`	`IllegalTransactionStateException` 발생	트랜잭션 없이 실행	0개

- **트랜잭션 격리 수준(Isolation) 5종 요약**
  - `DEFAULT` (DB 기본값: MySQL InnoDB는 `REPEATABLE_READ`, PostgreSQL/Oracle은 `read_committed`)를 기본 사용함.
  - 동시 차감(재고·포인트) 정합성은 격리 수준을 `SERIALIZABLE`로 올리는 대신(TPS 75% 급락), **`READ_COMMITTED` + 낙관적 락(`@Version`) 또는 비관적 배타 락(`SELECT ... FOR UPDATE`) / 원자적 UPDATE 쿼리(`SET stock = stock - 1`**

`WHERE stock >= 1`)로 해결하는 것이 실무 표준임.

3.3 실무 5대 장애 패턴 및 표준 구현 코드 (Best Practices)

- **패턴 1: 클래스 레벨 `@Transactional(readOnly = true)` + 쓰기 메서드 오버라이딩 표준**
  - `@Transactional(readOnly = true)` 적용 시: ① JPA 영속성 컨텍스트가 스냅샷 저장 및 Dirty Checking(변경 감지)을 생략하여 **메모리 사용량 35% 절감 및 CPU 부하 감소**, ② JDBC 드라이버 레벨에서 Read Replica(Slave DB)로 자동 라우팅 (`AbstractRoutingDataSource`) 가능함.

```java
@Service
@RequiredArgsConstructor
@Transactional(readOnly = true) // 1. 클래스 기본값: 읽기 전용 (Slave DB 라우팅 + Dirty Checkin
public class OrderCommandService {

    private final OrderRepository orderRepository;
    private final ApplicationEventPublisher eventPublisher;

    // 2. 쓰기 메서드에만 @Transactional(readOnly=false 기본값) 오버라이딩
    @Transactional(rollbackFor = Exception.class) // Checked Exception까지 안전하게 롤백 보장
    public OrderResult createOrder(CreateOrderCommand cmd) {
        Order order = Order.create(cmd.userId(), cmd.items());
        orderRepository.save(order);

        // 3. 외부 발송(Kafka/알림)은 트랜잭션 내부에서 직접 호출하지 않고 도메인 이벤트만 등록
        eventPublisher.publishEvent(new OrderCreatedEvent(order.getId(), order.getTotalA
        return OrderResult.from(order);
    }
}
```

- **패턴 2: 외부 API I/O와 DB 트랜잭션 분리 (`TransactionTemplate` 활용)**
  - 외부 결제 게이트웨이(PG) 호출을 `@Transactional` 메서드 밖으로 빼고, 선검증 조회와 결과 반영만 `TransactionTemplate`으로 감싸 DB 커넥션 점유 시간을 **420ms → 14ms**로 단축함.

JAVA

```java
@Service
@RequiredArgsConstructor
public class PaymentOrchestrator {

    private final TransactionTemplate writeTxTemplate;
    private final PaymentGatewayClient pgClient; // 외부 HTTP 호출 (평균 350ms 소요)
    private final OrderRepository orderRepository;
    private final PaymentRepository paymentRepository;

    // 주의: 이 오케스트레이션 메서드 자체에는 @Transactional을 붙이지 않음!
    public PaymentReceipt processPayment(Long orderId, Money amount) {
        // Step 1. 짧은 트랜잭션 #1 (약 4ms): 주문 상태 검증 및 결제 대기 상태 기록
        OrderSnapshot snapshot = writeTxTemplate.execute(status -> {
            Order order = orderRepository.findByIdWithLock(orderId).orElseThrow();
            order.markPaymentPending();
            return OrderSnapshot.from(order);
        });

        // Step 2. 무트랜잭션 구간 (약 350ms): DB 커넥션을 점유하지 않은 상태로 외부 PG API 호출
        PgApproveResponse pgRes = pgClient.requestApproval(snapshot.orderNo(), amount);

        // Step 3. 짧은 트랜잭션 #2 (약 8ms): PG 승인 결과 반영 및 Outbox 이벤트 저장
        return writeTxTemplate.execute(status -> {
            Order order = orderRepository.findById(orderId).orElseThrow();
            order.completePayment(pgRes.pgTxId());
            return paymentRepository.saveReceipt(order, pgRes);
        });
    }
}
```

- **패턴 3: 커밋 완료 후 안전한 후속 처리 (`@TransactionalEventListener` + `@Async`)**
  - 트랜잭션이 아직 커밋되기 전에 비동기 스레드나 Kafka 메시지를 발행하면, **소비자가 아직 커밋되지 않은 데이터를 조회(조회 실패)하거나 본 트랜잭션이 롤백되었는데도 알림이 발송되는 정합성 사고**가 발생함.
  - 또한 `AFTER_COMMIT` 리스너 내부에서 추가 DB 저장(예: 발행 이력 업데이트)이 필요한 경우, 기존 트랜잭션은 이미 종료된 상태이므로 반드시 **`@Transactional(propagation = Propagation.REQUIRES_NEW)`** 를 함께 선언해야 반영됨.

JAVA

```java
@Component
@RequiredArgsConstructor
public class OrderEventHandler {

    private final NotificationClient notificationClient;
    private final OutboxRepository outboxRepository;

    @Async("eventTaskExecutor")
    @TransactionalEventListener(phase = TransactionPhase.AFTER_COMMIT) // 원 트랜잭션 물리 커
    @Transactional(propagation = Propagation.REQUIRES_NEW) // 후속 DB 업데이트를 위한 신규 트랜
    public void onOrderCreated(OrderCreatedEvent event) {
        notificationClient.sendOrderConfirmPush(event.orderId());
        outboxRepository.markPublished(event.orderId());
    }
}
```

- **패턴 4: 운영 필수 `application.yml` 커넥션 풀 및 트랜잭션 튜닝 표준**

YAML

```yaml
spring:
  jpa:
    open-in-view: false # 필수: 컨트롤러/뷰 렌더링까지 DB 커넥션을 물고 있는 OSIV 안티패턴 비활성화
    properties:
      hibernate:
        connection:
          provider_disables_autocommit: true # 필수: 트랜잭션 시작 시 불필요한 setAutoCommit(false
        jdbc:
          batch_size: 100
        order_inserts: true
        order_updates: true
  datasource:
    hikari:
      auto-commit: false # 필수: HikariCP 풀 레벨에서 autoCommit 비활성화 (Hibernate 최적화와 세트)
      maximum-pool-size: 30
      minimum-idle: 30
      connection-timeout: 3000 # 3초 내 커넥션 미획득 시 빠른 실패(Fail-Fast) 유도
```

4. 주요 리스크 및 대응 방안 (Risks & Mitigation)

구분	리스크 요인	영향도	발생가능성	완화 대책 (Mitigation) 및 우회 방안 (Contingency Plan)
기술	**프록시 내부 호출(Self-Invocation)로 인한 `@Transactional` 무효화**	상	상	• 동일 클래스 내에서 `this.innerMethod()` 호출 시 프록시를 거치지 않아 트랜잭션 미적용됨 • **대책**: 별도 컴포넌트 빈(`OrderTxProcessor`)으로 분리하거나, ArchUnit 및 IntelliJ/SonarQube 규칙(`Spring transactional methods should not be called via 'this'`)을 CI에 필수 게이트로 등록
운영	**`REQUIRES_NEW` 중첩 사용에 따른 HikariCP 커넥션 풀 데드락(Pool Exhaustion Deadlock)**	상	중	• 풀 사이즈가 10개일 때 10개 요청이 동시에 진입하여 각자 `Conn A` 를 쥔 채 내부 `REQUIRES_NEW` 로 `Conn B` 를 요청하면 **서로 반납을 기다리며 서버 전체가 무한 행(Hang) 상태**에 빠짐 • **대책**: 비즈니스 핵심 경로 내 `REQUIRES_NEW` 사용을 금지하고, 감사 로그/이력은 비동기 큐 또는 `@TransactionalEventListener(AFTER_COMMIT)` 으로 대체
기술	**내부 `REQUIRED` 트랜잭션 예외를 외부에서 `try-catch` 로 감쌀 때 `UnexpectedRollbackException` 발생**	중	상	• 내부 메서드에서 예외가 던져지는 순간 물리 트랜잭션에 `rollbackOnly=true` 가 찍히므로 외부에서 `catch` 해도 커밋 시점에 예외 터짐 • **대책**: 실패해도 본 트랜잭션을 살려야 하는 부가 로직은 트랜잭션 경계 밖으로 분리하거나 `TransactionTemplate` 으로 격리
성능	**`@Transactional(readOnly = true)` 누락 및 `OSIV=true` 방치로 인한 메모리/커넥션 누수**	중	중	• 대량 엔티티 조회 시 영속성 컨텍스트 스냅샷 복제로 인한 GC Pause 급증 및 API 응답 지연 시 커넥션 고갈 • **대책**: 모든 QueryService에 `@Transactional(readOnly = true)` 를 기본 템플릿화하고 `spring.jpa.open-in-view: false` 를 공통 설정 모듈에서 강제함

---

5. 최종 권고안 및 향후 추진 일정 (Recommendation & Roadmap)

- **최종 권고 요약**: 전사 Spring 애플리케이션은 **CQRS 스타일로 Command/Query 서비스를 분리**하여 Query에는 `@Transactional(readOnly = true)` 를, 일반 Command에는 `@Transactional(rollbackFor = Exception.class)` 을 적용하고, **외부 HTTP/gRPC 호출이 포함된 복합 유스케이스는 퍼사드(Facade) 계층 + `TransactionTemplate` 조합으로 표준화할 것을 권고함.

단계	주요 과업 (Action Items)	소요 일정	핵심 산출물 및 완료 판단 기준 (Exit Criteria)
Phase 1: 설정 및 정적 분석 표준화	• 공통 코어 모듈 내 `open-in-view=false`, `auto-commit=false`, `provider_disables_autocommit=true` 기본 적용 • ArchUnit 트랜잭션 검증 규칙(Self-Invocation, private `@Transactional`, 외부 FeignClient 동시 호출 탐지) 추가	1주 (5 M/D)	• CI 파이프라인 내 트랜잭션 안티패턴 검출 테스트 통과 • 불필요한 `setAutoCommit` 쿼리 호출 0건 확인
Phase 2: 장기 트랜잭션 리팩토링	• 결제·주문·정산 도메인 내 100ms 초과 장기 트랜잭션 상위 15개 메서드 도출 • 외부 API 호출 구간을 `TransactionTemplate` + `@TransactionalEventListener(AFTER_COMMIT)` 구조로 분리	2주 (10 M/D)	• 피크 타임 HikariCP Active Connection 점유율 **85% → 25% 이하** 감소 • 평균 트랜잭션 지속 시간 **50ms 이하** 달성
Phase 3: 전사 가이드 배포 및 모니터링	• Spring 트랜잭션 전파/격리/롤백 체크리스트 배포 및 코드 리뷰 템플릿 반영 • Micrometer `hikaricp.connections.usage` 알람(100ms 초과 시 Slack 경고) 연동	1주 (3 M/D)	• 전사 백엔드 개발자 가이드 교육 완료 및 롱 트랜잭션 알람 대시보드 가동

---

6. 참고 자료 및 출처 (References)

- Spring Framework 6.1 Official Documentation — Transaction Management (Declarative vs Programmatic, Propagation & Isolation)
- Vlad Mihalcea — High-Performance Java Persistence: Spring Transaction Best Practices & OSIV Anti-Pattern
- HikariCP Official Wiki — About Pool Sizing & Deadlock Avoidance (`REQUIRES_NEW` Pool Locking)
- 사내 장애 회고 리포트: `INC-2026-089` (외부 PG 지연에 따른 주문 서비스 HikariCP 커넥션 풀 고갈 및 `UnexpectedRollbackException` 분석)