Django transaction

먼저 장고의 기본 트랜잭션 모델을 마음속에 그려 둬야 한다. 장고는 기본적으로 오토커밋 모드다. ORM이 INSERT·UPDATE·DELETE를 실행하는 순간 데이터베이스는 즉시 커밋한다. 이 상태에서는 하나의 요청 안에서 여러 쓰기 작업이 서로의 성공 여부에 묶이지 않는다. 반대로 특정 구간을 원자적으로 묶고 싶다면 transaction.atomic()으로 오토커밋을 잠시 꺼, 그 블록이 전부 성공하면 커밋하고 하나라도 실패하면 롤백하게 만든다. 이때 장고는 내부적으로 세이브포인트를 사용하여 중첩 atomic()도 허용한다. 바깥 블록은 전체 트랜잭션의 시작과 끝을, 안쪽 블록은 저장점 수준의 부분 롤백을 제공한다. 이 구조를 이해하면 "크게 한 번, 안에서 몇 번"이라는 전략을 세울 수 있다. 예컨대 결제 승인과 주문 상태 전이를 하나의 큰 atomic으로 묶되, 중간에 주소 정규화 같은 부수 작업은 작은 저장점 안에서 실패 시 해당 부분만 되돌리고 나머지는 이어 가는 식의 미세 조정이 가능해진다. 전역적으로 모든 뷰를 자동으로 atomic으로 감싸는 ATOMIC_REQUESTS 설정도 있지만, 요청 전 과정을 트랜잭션으로 묶는 것은 장기 락, 교착상태, 불필요한 롤백 전파 같은 부작용을 초대하기 쉽다. 실무에서는 유스케이스 경계를 기준으로 필요한 곳에만 명시적으로 atomic()을 두고, 그 외에는 오토커밋의 단순함을 유지하는 편이 안전하다.

트랜잭션의 목적은 ACID를 지키는 것이다. 원자성은 말 그대로 "전부 아니면 전무"이고, 일관성은 도메인 불변식을 깨지 않는다는 뜻이며, 고립성은 동시 실행 중 서로의 중간 상태가 관찰되지 않게 하는 약속이고, 지속성은 커밋된 변경이 영구히 남는 성질이다. 장고 차원에서 우리가 직접 제어할 수 있는 것은 원자성 경계와 고립성의 일부다. 원자성은 atomic()으로, 고립성은 데이터베이스의 격리 수준과 락으로 다룬다. PostgreSQL의 기본 격리는 READ COMMITTED라서 각 문이 실행될 당시 커밋된 데이터만 본다. 이 수준에서도 경쟁 조건은 얼마든지 발생한다. 재고를 두 클라이언트가 동시에 차감하거나, 같은 고유키를 만드는 요청이 거의 동시에 도착하면, 둘 다 통과해 중복 데이터나 음수 재고가 생길 수 있다. 이런 상황을 소프트웨어 설계만으로 막을 수 있다고 믿는 것은 낭만이다. 도메인 불변식이 중요할수록 데이터베이스 락과 제약조건을 끌어들여야 한다.

이때 가장 실용적인 도구가 select_for_update()다. 이는 조회된 행을 **행 락(row-level lock)**으로 잡아 같은 행을 갱신하려는 다른 트랜잭션을 대기시키거나 실패시키게 한다. 예를 들어 "재고가 1 이상이어야 차감 가능"이라는 규칙을 코드 앞단에서 검사하는 것만으로는 부족하다. 두 요청이 거의 동시에 들어오면 둘 다 stock >= 1을 보고 통과해, 결과적으로 재고가 음수가 될 수 있다. atomic() 블록 안에서 Product.objects.select_for_update().get(id=...)로 행을 잠그고, 그 안에서 다시 검사하고 차감하면, 두 번째 트랜잭션은 첫 번째가 커밋하거나 롤백할 때까지 대기한다. PostgreSQL이라면 nowait=True나 skip_locked=True 옵션으로 대기 대신 즉시 실패하거나 건너뛰는 전략도 취할 수 있다. 후자는 작업 큐에서 작업 예약을 구현할 때 빛난다. "잠겨 있지 않은 선두 작업을 잡아서 처리하고, 잠겨 있으면 건너뛰고 다음으로"라는 패턴이 select_for_update(skip_locked=True) 한 줄로 안정적으로 구현된다. 반대로, F() 표현식으로 원자적 업데이트를

만드는 것도 경쟁 조건을 줄이는 데 효과적이다. update(balance=F('balance') - amount) 같은 식은 조회→계산→저장 사이의 공백을 줄여준다. 다만 불변식이 복잡할수록 행 락과 제약조건이 더 신뢰할 수 있는 방패다.

여기서 고유 제약과 업서트가 등장한다. "같은 외부 이벤트 ID로는 한 번만 반영" 같은 요구는 애플리케이션 레벨의 멱등키 저장만으로는 모자랄 때가 있다. 테이블에 고유 인덱스를 두고, 삽입 시 IntegrityError가 터지면 중복으로 본 뒤 안전하게 무시하는 전략은 단순하면서 강력하다. 장고의 get_or_create()와 update_or_create()는 이 패턴을 얇게 감싸 준다. 다만 완벽한 멀티샷 안전성을 원한다면 제약의 검사 시점도 챙겨야 한다. PostgreSQL은 제약을 DEFERRED로 미루어 커밋 시에만 검사하게 할 수 있다. 경쟁적으로 같은 키를 만들 수밖에 없는 복잡한 생성 시나리오에서는 제약을 지연시키고, 마지막 커밋 단계에서 한 번만 충돌을 해소하는 설계를 고려할 수 있다. 반대로 MySQL의 InnoDB는 일부 DDL이 암묵 커밋을 일으키거나 제약 처리 동작이 달라 "한 DB에서 된다고 다른 DB에서도 된다"는 가정을 금지한다. 장고는 데이터베이스 별로 미묘한 차이를 흡수해 주지만, 트랜잭션과 제약의 가장자리는 결국 DB의 땅이다.

실무에서 트랜잭션의 경계를 정할 때 가장 자주 부딪히는 딜레마는 부수효과의 시점이다. 이메일 발송, 웹훅 호출, 메시지 큐 퍼블리시 같은 외부 상호작용을 트랜잭션 안에서 해버리면 어떤 일이 생길까. DB 변경이 나중에 롤백되더라도 외부 세계는 이미 알림을 받았거나 과금이 일어났을 수 있다. 반대로 트랜잭션 밖에서 하자니 커밋이 실패할 수도 있는데, 실패한 사실을 모른 채 외부 효과만 살아남을 수도 있다. 이때 장고의 transaction.on_commit() 콜백이 결정적이다. 이 함수에 콜백을 등록하면 커밋이 실제로 성공한 이후에만 호출된다. 외부 알림, Celery 태스크 enqueue, 캐시 무효화 같은 일은 되도록 on_commit() 내부로 보내라. 그러면 롤백 시에는 콜백이 실행되지 않으니 데이터와 부수효과의 일관성을 확보할 수 있다. 더 나아가 "DB에 이벤트를 먼저 안전하게 기록하고, 커밋 후 그 이벤트를 발행"하는 아웃박스 패턴을 적용하면, 메시징까지 포함한 강한 일관성을 확보한다. 장고 모델로 Outbox 테이블을 만들고, 비즈니스 트랜잭션 안에서 이벤트 행을 함께 INSERT한 뒤, on_commit()으로 퍼블리셔를 깨워 Outbox를 비워나가면, 다운타임이나 재시작에도 이벤트 유실을 막을 수 있다.

트랜잭션에서 예외를 다루는 태도도 중요하다. DB 드라이버가 IntegrityError나 OperationalError를 던지면 장고는 현재 트랜잭션을 에러 상태로 표시한다. 이 상태에서 추가 DB 작업을 시도하면 TransactionManagementError가 터진다. 즉, 한 번 실패한 트랜잭션 안에서는 더 이상의 DB 호출을 하지 말고, 바로 빠져나가라. atomic() 블록은 문맥 관리자로서 예외 전파 시 자동으로 롤백하고, 블록 바깥으로 나가면 상태를 정상으로 되돌려 준다. 특별한 이유가 없다면 set_rollback()을 직접 만질 일은 드물다. 반대로 일시적인 락 타임아웃이나 데드락은 재시도가 해법이다. select_for_update(nowait=True)로 즉시 실패하게 설계했으면, 실패를 캐치해 10~50ms 사이 무작위 지연 후 한두 번 재시도하는 전략을 권한다. 무한 재시도는 장애를 키우는 지름길이다. 중요한 것은, 재시도가 안전하려면 연산 자체가 멱등해야 한다는 점이다. 멱등키나 고유 제약을 통해 재시도를 무해하게 만들어 두지 않으면, 재시도는 중복 반영의 다른 이름이 된다.

트랜잭션은 길수록 위험하다. 길다는 것은 DB 락을 오래 잡는다는 뜻이고, 이는 다른 트랜잭션의 대기, 더

나아가 교착상태의 씨앗이 된다. 또한 PostgreSQL에서는 트랜잭션이 열린 채로 오래 머무르면 **MVCC 히스토리 청소(autovacuum)**가 지연되어 테이블과 인덱스가 비대해지고 전체 성능이 악화된다. 그러므로 트랜잭션 안에서는 네트워크 호출, 파일 I/O, 장시간 CPU 연산을 피하라. 입력 검증, 외부 서비스 조회, 이미지 리사이즈 같은 일은 트랜잭션 밖에서 선행하고, 오직 "상태를 바꾸는 최소한의 DB 쓰기"만 짧고 굵게 묶어라. API에서 스트리밍 응답을 생성하는 동안 트랜잭션을 열어 두는 실수도 잦다. 스트리밍은 응답이 끝날 때까지 연결을 유지하므로, 그동안 락이 유지되어 시스템 전체가 둔해진다. 스트리밍은 트랜잭션과 철저히 분리하라.

복수 데이터베이스 환경에서는 난도가 더 올라간다. 장고는 atomic(using="alias")로 DB별 트랜잭션 경계를 따로 지정하게 한다. 두 개의 DB에 걸쳐 하나의 논리 트랜잭션을 만들고 싶다는 욕망은 당연하지만, 장고는 **분산 트랜잭션(2PC)**을 제공하지 않는다. 현실적인 해법은 "중요한 쓰기는 한 DB에 몰고, 다른 DB로는 비동기 복제나 보조 데이터만 둔다"는 설계다. 읽기 부하 분산을 위해 리드 레플리카를 두는 경우, 쓰기 직후 읽기를 레플리카로 보내면 복제 지연으로 직전 커밋이 보이지 않는 일이 생긴다. 이때는 "쓰기 직후 특정 읽기만 마스터로 강제"하거나, 요청 컨텍스트에 일관성 플래그를 달아 중요 조회는 마스터에서, 덜 중요한 조회는 레플리카에서 하게 분기하는 식의 정책을 세워야 한다. 장고의 데이터베이스 라우터로 이를 부분 자동화할 수 있지만, 트랜잭션의 경계가 어디인지, 어느 읽기가 강한 일관성을 요구하는지에 대한 도메인 지식이 먼저다.

테스트에서도 트랜잭션의 성질을 정확히 이해해야 덜 고생한다. 장고의 TestCase는 각 테스트를 트랜잭션으로 감싸고 끝나면 롤백해 DB를 깨끗이 돌려준다. 이 편의성 뒤에는 한 가지 함정이 숨어 있는데, 바로 on_commit() 콜백이 기본적으로 실행된다는 점이다. 즉, 트랜잭션 내부지만 테스트의 끝에서 커밋을 흉내 내며 on_commit()을 수행한다. 반대로 트랜잭션 동작 자체(락, 데드락, 커밋 지연)를 검증해야 한다면 TransactionTestCase를 써라. 이 클래스는 테스트마다 실제 커밋/롤백을 발생시키므로 DB의 진짜 동작을 재현할 수 있다. 락을 테스트하려면 두 스레드(혹은 프로세스)에서 같은 행을 select_for_update로 잡아보는 식으로 시뮬레이션을 걸 수 있다. 또한 API 엔드포인트의 성능 회귀를 막으려면 "이 요청은 상수 개의 쿼리만 수행해야 한다"는 단언을 추가하라. 트랜잭션 경계가 무너져 프리페치가 빠졌을 때, 쿼리 수가 급증하는 것을 테스트가 즉시 잡아준다.

실전에서 자주 마주치는 세 가지 레시피를 사례로 살펴보자. 첫째, 돈 이체다. 같은 계좌에서 출금과 입금은 반드시 함께 성공하거나 함께 실패해야 한다. 서비스 레이어의 메서드 하나를 atomic()으로 묶고, 출금 계좌와 입금 계좌를 항상 같은 순서로 select_for_update()로 잠가 교착을 피한다. 그 안에서 잔액을 검사하고 F() 연산으로 차감·증가를 수행하며, 부정 시나리오에서는 예외를 던져 전체 롤백을 유도한다. 외부 알림(푸시·메일)과 회계 로그 발행은 on_commit()으로 미룬다. 둘째, 웹훅 멱등성이다. 외부 결제사가 같은 이벤트를 두 번 보낼 수 있다. 이벤트 ID에 고유 제약을 둔 테이블에 먼저 원자적 삽입을 시도하고, 성공하면 비즈니스 반영을 이어가며, 실패하면 중복으로 간주해 안전하게 무시한다. 이 모든 과정은 하나의 atomic()으로 묶인다. 반영 후 외부에 후속 호출이 필요하다면 역시 on_commit()을 쓴다. 셋째, 작업 큐에서 예약이다. 워커는 "대기 중인 작업 중 하나를 집어서 내 것으로 만들고 처리"해야 한다. 이때 select_for_update(skip_locked=True)

를 사용해 "남이 잡은 작업은 건너뛰는" 쿼리를 만들면, 다수 워커가 동시에 뛰어도 서로 방해하지 않고 자연스레 분산된다. 처리 결과 저장과 상태 전이는 같은 트랜잭션 안에서 일어나야 하며, 외부 사이드이펙트는 커밋 후에만 실행된다.

트랜잭션은 모델링과도 맞물린다. 도메인의 불변식이 모델 내부에서 강제될수록, 트랜잭션은 짧고 단단해진다. 예컨대 주문이 "PAID가 아니면 FULFILLED가 될 수 없다"는 규칙을 Order.fulfill() 메서드에서 검사하면, 서비스는 유스케이스를 조립하면서 atomic()만 걸고 메서드를 호출하면 된다. 불변식이 흩어져 뷰·시리얼라이저·서비스에 나눠져 있으면, 아무리 트랜잭션을 잘 묶어도 누수가 생긴다. 반대로 불변식을 모델에 몰아두면, 트랜잭션 경계가 단순해지고 테스트도 단위별로 깔끔해진다. 이때 데이터베이스 제약(UNIQUE, CHECK, FK)과 모델의 검증(clean, full_clean)을 중복으로 두는 것을 두려워하지 마라. 서로 다른 층에서 같은 규칙을 보강하면, 성능과 안전성의 균형을 잡을 수 있다. DB 제약은 마지막 방어선이고, 모델 검증은 사용자 경험과 오류 메시지의 품질을 좌우한다.

트랜잭션과 비동기의 접점도 짚고 넘어가자. 장고 ORM은 동기 기반이다. ASGI 환경에서 async def 뷰를 쓰더라도, ORM 호출은 스레드 풀에서 실행된다. 중요한 것은 트랜잭션 경계를 스레드 경계 밖으로 흘리지 않는 것이다. atomic() 블록 안에서 비동기 대기를 길게 할수록 락을 오래 잡고 있다는 뜻이다. DB 쓰기와 직결되지 않는 기다림(외부 API, 파일 업로드 등)은 최대한 트랜잭션 밖에서 해결하고, "잠깐의 쓰기"만 안으로 들여라. Celery 같은 작업 큐를 통해 아예 후행 작업을 뗄 때는, 큐에 태스크를 넣는 행위 자체가 커밋 후에만 일어나도록 반드시 on_commit()을 붙여라. 그렇지 않으면 롤백된 데이터에 의존한 태스크가 실행되어 시스템 외부로 오류가 증폭된다.

끝으로, 트랜잭션은 철학이 아니다. 운영의 학문이다. 로그와 메트릭은 "얼마나 자주 롤백이 일어나는지", "락 대기 시간과 데드락 빈도는 어떤지", "장기 트랜잭션이 얼마나 발생하는지"를 보여주는 계기판이다. 커넥션 풀의 크기, 타임아웃 설정, deadlock_timeout이나 lock_timeout 같은 DB 파라미터는 트랜잭션 전략의 외부 환경이다. 짧고 명확한 경계, 락을 이해한 쿼리, 커밋 후로 미루는 부수효과, 멱등성과 재시도, DB 제약을 활용한 최후의 방어선, 이 다섯 가지만 지켜도 실무의 대부분 사고는 초기에 차단된다. 장고는 이 모든 것을 구현하기 위한 훌륭한 손잡이들을 제공한다. atomic()은 경계를, select_for_update()는 고립성을, F()와 고유 제약은 경쟁 안전을, on_commit()은 외부 세계와의 일관성을, 그리고 테스트 도구들은 회귀를 막는 안전망을 제공한다. 결국 트랜잭션은 코드 몇 줄이 아니라 태도다. "무엇을 함께 성공시키고, 무엇을 함께 실패시킬지"를 먼저 정하고, 그 다음에 장고의 도구들로 그 경계를 정확히 그려라. 그러면 서비스는 예측 가능해지고, 성능은 안정되며, 팀은 두려움 없이 변경할 수 있다.