4. 데이터 변경 이력 설계1

#0.강의/2.데이터베이스로드맵/4.설계2

- /데이터 변경 이력 설계가 필요한 이유
- /변경 추적 컬럼 - 기본
- /변경 추적 컬럼 - 변경 사유
- /변경 추적 컬럼 - 감사(Audit) 컬럼
- /정리

데이터 변경 이력 설계가 필요한 이유

학습 방향

- **데이터 변경 이력 설계1**: 변경 추적 컬럼
- **데이터 변경 이력 설계2**: 이력 관리

데이터 변경 이력 설계가 필요한 이유

데이터베이스를 설계할 때 많은 개발자들이 간과하는 부분이 있다. 바로 "데이터의 변경 이력"이다. 처음 서비스를 개발할 때는 현재 데이터만 잘 저장하면 된다고 생각한다. 하지만 서비스가 성장하고 시간이 지나면 반드시 이런 질문들이 나온다.

- "이 상품의 가격이 원래 얼마였죠?"
- "고객이 주소를 언제 변경했나요?"
- "이 데이터를 누가 수정했나요?"
- "작년 이맘때 회원 등급이 어떻게 되어 있었나요?"

이런 질문에 답할 수 없다면 심각한 문제가 발생한다. 고객 클레임을 처리할 수 없고, 감사(Audit)에 대응할 수 없으며, 데이터 오류가 발생해도 원인을 추적할 수 없다.

실무에서는 다음과 같은 상황이 빈번하게 발생한다.

사례 1: 가격 분쟁
고객이 "저는 분명히 10,000원에 주문했는데 왜 12,000원이 결제되었나요?"라고 문의한다. 상품 테이블을 확인하니 현재 가격은 12,000원이다. 하지만 고객이 주문한 시점의 가격이 얼마였는지 알 수 없다. 가격 변경 이력이 없기 때문이다.

사례 2: 배송 주소 오류

배송이 반송되었다. 고객은 "주소를 정확하게 입력했다"고 주장한다. 하지만 현재 주소는 다른 주소로 되어 있다. 고객이 주소를 변경한 것인지, 시스템 오류인지 알 수 없다. 주소 변경 이력이 없기 때문이다.

사례 3: 감사 대응

금융 서비스에서 감사관이 "이 고객의 한도가 언제, 누구에 의해, 왜 변경되었는지 보여주세요"라고 요청한다. 현재 한도만 알 수 있고, 변경 이력이 없다면 심각한 규정 위반이 된다.

이처럼 데이터 변경 이력은 선택이 아니라 필수다. 이번 수업에서는 데이터 변경 이력을 관리하는 다양한 방법을 단계별로 알아보겠다. 가장 단순한 방법부터 시작해서 실무에서 가장 많이 사용하는 패턴까지 하나씩 발전시켜 보자.

이력이 없는 원본 테이블

먼저 아무런 이력 관리도 하지 않는 가장 기본적인 테이블 구조를 살펴보자. 쇼핑몰에서 상품을 관리하는 테이블이다.

```
DROP TABLE IF EXISTS product;

CREATE TABLE product (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(200) NOT NULL,
    price INT NOT NULL,
    stock_quantity INT NOT NULL DEFAULT 0,
    status VARCHAR(20) NOT NULL DEFAULT 'ACTIVE'
);
```

상품 데이터를 등록해보자.

```
INSERT INTO product (name, price, stock_quantity, status)
VALUES ('스마트폰 케이스', 15000, 100, 'ACTIVE');

INSERT INTO product (name, price, stock_quantity, status)
VALUES ('무선 이어폰', 89000, 50, 'ACTIVE');

INSERT INTO product (name, price, stock_quantity, status)
VALUES ('노트북 파우치', 35000, 30, 'ACTIVE');
```

현재 상품 목록을 확인해보자.

```
SELECT * FROM product;
```

[실행 결과]

product_id	name	price	stock_quantity	status
1	스마트폰 케이스	15000	100	ACTIVE
2	무선 이어폰	89000	50	ACTIVE
3	노트북 파우치	35000	30	ACTIVE

데이터가 잘 저장되어 있다. 이제 15,000원인 스마트폰 케이스 상품 가격을 12,000원으로 변경해보자.

```
UPDATE product
SET price = 12000
WHERE product_id = 1;
```

변경 후 데이터를 확인해보자.

```
SELECT * FROM product WHERE product_id = 1;
```

[실행 결과]

product_id	name	price	stock_quantity	status
1	스마트폰 케이스	12000	100	ACTIVE

가격이 15,000원에서 12,000원으로 변경되었다. 하지만 여기서 문제가 발생한다.

문제점 분석

이 테이블 구조에서는 다음 질문에 전혀 답할 수 없다.

- 질문 1: 누가 변경했나요?
- 질문 2: 언제 변경되었나요?
- 질문 3: 왜 변경되었나요?
- 질문 4: 이전 값은 얼마였나요?

이처럼 아무런 이력 관리를 하지 않으면 데이터가 변경된 순간 과거의 모든 정보가 사라진다. 이것은 마치 연필로 쓴 글씨를 지우개로 지우고 새로 쓰는 것과 같다. 지워진 글씨는 영원히 알 수 없다.

실무에서 발생하는 심각한 문제

실무에서 이런 구조로 서비스를 운영하면 다음과 같은 심각한 문제가 발생한다.

문제 상황 시뮬레이션

고객이 어제 주문을 했다. 당시 가격은 15,000원이었다. 하지만 오늘 가격이 12,000원으로 변경되었다.

```
-- 고객의 주문 내역 (주문 테이블이 있다고 가정)
-- 주문 시점: 어제
-- 주문 당시 상품 가격: ???

-- 현재 상품 테이블에서 조회
SELECT product_id, name, price AS current_price
FROM product
WHERE product_id = 1;
```

[실행 결과]

product_id	name	current_price
1	스마트폰 케이스	12000

오늘은 현재 가격 12,000원만 알 수 있다. 고객이 주문한 어제 시점의 가격 15,000원은 이미 사라졌다. 만약 고객이 "주문할 때 가격이 다르게 표시되었다"고 클레임을 제기하면, 누가 언제 가격을 변경했는지, 그리고 과거의 가격이

15000원이었는지 이를 확인할 방법이 전혀 없다.

이러한 문제를 해결하기 위해 변경 이력을 관리하는 다양한 방법을 알아보겠다. 가장 단순한 방법부터 시작해서 점점 발전된 방법으로 나아갈 것이다.

변경 추적 컬럼 - 기본

가장 먼저 적용할 수 있는 간단한 방법은 테이블에 변경 추적 컬럼을 추가하는 것이다. 누가, 언제 데이터를 등록하고 수정했는지 기록하는 컬럼이다.

변경 추적 컬럼 추가

기존 테이블을 삭제하고 변경 추적 컬럼이 추가된 새 테이블을 생성하자.

```
DROP TABLE IF EXISTS product;

CREATE TABLE product (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(200) NOT NULL,
    price INT NOT NULL,
    stock_quantity INT NOT NULL DEFAULT 0,
    status VARCHAR(20) NOT NULL DEFAULT 'ACTIVE',
    -- 변경 추적 컬럼
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    created_by VARCHAR(100) NOT NULL,
    updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    updated_by VARCHAR(100) NOT NULL
);
```

각 컬럼의 의미는 다음과 같다.

- `created_at` : 데이터가 처음 등록된 시간
- `created_by` : 데이터를 처음 등록한 사람 (또는 시스템)
- `updated_at` : 데이터가 마지막으로 수정된 시간

- `updated_by` : 데이터를 마지막으로 수정한 사람 (또는 시스템)

데이터 등록

이제 데이터를 등록할 때 누가 등록했는지 함께 기록하자.

```
INSERT INTO product (name, price, stock_quantity, status, created_by,
updated_by)
VALUES ('스마트폰 케이스', 15000, 100, 'ACTIVE', 'admin_kim', 'admin_kim');

INSERT INTO product (name, price, stock_quantity, status, created_by,
updated_by)
VALUES ('무선 이어폰', 89000, 50, 'ACTIVE', 'admin_lee', 'admin_lee');

INSERT INTO product (name, price, stock_quantity, status, created_by,
updated_by)
VALUES ('노트북 파우치', 35000, 30, 'ACTIVE', 'admin_kim', 'admin_kim');
```

등록된 데이터를 확인해보자.

```
SELECT product_id, name, price, created_at, created_by, updated_at, updated_by
FROM product;
```

[실행 결과]

product_id	name	price	created_at	created_by	updated_at	upd
1	스마트폰 케이스	15000	2026-01-15 10:00:00	admin_kim	2026-01-15 10:00:00	adr
2	무선 이어폰	89000	2026-01-15 10:05:00	admin_lee	2026-01-15 10:05:00	adr
3	노트북 파우치	35000	2026-01-15 10:10:00	admin_kim	2026-01-15 10:10:00	adr

이제 누가, 언제 데이터를 등록했는지 알 수 있다.

참고로 상품이 최초로 등록되어도 수정일을 보통 함께 저장해둔다. 왜냐하면 데이터를 조회하거나 정렬하는 로직을 단순화하기 위해서다.

만약 등록 시점에 수정일(`updated_at`)을 `NULL`로 비워둔다면, "가장 최근에 변경된 순서"로 데이터를 조회할 때 쿼리가 복잡해진다. 수정일이 없으면 생성일을 참조해야 하므로 `COALESCE(updated_at, created_at)` 같은 함수를 매번 써야 하기 때문이다. 생성도 일종의 '첫 번째 수정' 상태로 간주하여 값을 채워넣는 것이 개발 편의성 면에서 훨씬 유리하다.

데이터 수정

상품 가격을 수정해보자.

이제부터 수정할 때는 누가 수정했는지도 함께 업데이트해야 한다.

```sql
UPDATE product
SET price = 12000,
    updated_by = 'admin_park'
WHERE product_id = 1;
```

수정된 데이터를 확인해보자.

```sql
SELECT product_id, name, price, created_at, created_by, updated_at, updated_by
FROM product
WHERE product_id = 1;
```

[실행 결과]

product_id	name	price	created_at	created_by	updated_at	upd
1	스마트폰 케이스	12000	2026-01-15 10:00:00	admin_kim	2026-01-16 14:30:00	adr

이제 다음 질문에 답할 수 있다.

- 언제 등록되었나요? → 2026-01-15 10:00:00
- 누가 등록했나요? → admin_kim

- 언제 수정되었나요? → 2026-01-16 14:30:00
- 누가 수정했나요? → admin_park

장점

변경 추적 컬럼을 사용하면 다음과 같은 장점이 있다.

1. **설계가 단순하다**: 기존 테이블에 4개의 컬럼만 추가하면 된다.
2. **애플리케이션 코드가 간단하다**: 등록과 수정 시 해당 컬럼만 채워주면 된다.

남은 문제

- **질문 1: 누가 변경했나요? ➜ 해결**
- **질문 2: 언제 변경되었나요? ➜ 해결**
- 질문 3: 왜 변경되었나요? ➜ 미해결
- 질문 4: 이전 값은 얼마였나요? ➜ 미해결

이렇게 해서 누가, 언제 데이터를 변경했는지는 알 수 있게 되었다. 그런데 아직까지 왜 변경되었는지와, 변경 이전의 값을 알 수 없다.

언제 사용하면 좋을까?

단순히 등록일, 수정일, 등록자, 수정자 컬럼만 추가하는 이 방식은 다음과 같은 경우에 적합하다.

1. **이력이 중요하지 않은 마스터 데이터**: 코드 테이블, 설정 테이블 등
2. **단순 운영 도구**: 내부 관리용 데이터
3. **어드민 로그 정도의 용도**: 누가 마지막으로 수정했는지만 알면 되는 경우

하지만 가격 변경, 주문 상태 변경, 회원 등급 변경 등 이력이 중요한 데이터에는 이 방식만으로는 부족하다.

다음 시간에는 질문 3: 왜 변경되었는지, 변경 사유를 해결해보자.

> **실무 이야기**
> 실무에서 변경 추적 컬럼은 거의 모든 테이블에 다 적용한다.
> 그래서 실무의 테이블은 대부분 등록일, 등록자, 수정일, 수정자 컬럼을 포함한다.
> 데이터가 절대 변하면 안 되는 테이블의 경우에는 등록일, 등록자 컬럼만 포함하면 된다.

변경 추적 컬럼 - 변경 사유

앞서 배운 기본 변경 추적 컬럼으로는 "누가, 언제" 수정했는지는 알 수 있지만, "왜" 수정했는지는 알 수 없다. 실무에서는 변경 사유가 중요한 경우가 많다.

문제 상황

상품 가격이 변경되었다. 담당자에게 물어보니 "할인 이벤트 때문에 변경했습니다"라고 한다. 하지만 담당자가 퇴사하거나, 시간이 오래 지나면 왜 변경했는지 알 수 없게 된다.

- "이 가격 변경은 할인 이벤트인가요, 원가 조정인가요?"
- "언제까지 유효한 가격인가요?"
- "누구의 승인을 받고 변경한 건가요?"

이런 질문에 답하려면 변경 사유를 기록해야 한다.

변경 사유 컬럼 추가

테이블에 변경 유형과 변경 사유를 기록하는 컬럼을 추가하자.

```
DROP TABLE IF EXISTS product;

CREATE TABLE product (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(200) NOT NULL,
    price INT NOT NULL,
    stock_quantity INT NOT NULL DEFAULT 0,
    status VARCHAR(20) NOT NULL DEFAULT 'ACTIVE',
    -- 기본 변경 추적 컬럼
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    created_by VARCHAR(100) NOT NULL,
    updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    updated_by VARCHAR(100) NOT NULL,
    -- 변경 사유 컬럼
```

```
    change_type VARCHAR(50),
    change_reason VARCHAR(500)
);
```

추가된 컬럼의 의미는 다음과 같다.

- `change_type` : 변경 유형 (예: PRICE_CHANGE, STATUS_CHANGE, STOCK_ADJUST 등)
- `change_reason` : 변경 사유 (자유 형식의 텍스트)

데이터 등록

데이터를 등록할 때는 변경 유형을 'CREATE'로 기록하자.

```
INSERT INTO product (name, price, stock_quantity, status, created_by,
updated_by, change_type, change_reason)
VALUES ('스마트폰 케이스', 15000, 100, 'ACTIVE', 'admin_kim', 'admin_kim',
'CREATE', '신규 상품 등록');

INSERT INTO product (name, price, stock_quantity, status, created_by,
updated_by, change_type, change_reason)
VALUES ('무선 이어폰', 89000, 50, 'ACTIVE', 'admin_lee', 'admin_lee', 'CREATE',
'신규 상품 등록');
```

```
SELECT product_id, name, price, change_type, change_reason, updated_by
FROM product;
```

[실행 결과]

product_id	name	price	change_type	change_reason	updated_by
1	스마트폰 케이스	15000	CREATE	신규 상품 등록	admin_kim
2	무선 이어폰	89000	CREATE	신규 상품 등록	admin_lee

가격 변경 - 할인 이벤트

할인 이벤트로 가격을 변경해보자.

```sql
UPDATE product
SET price = 12000,
    updated_by = 'admin_park',
    change_type = 'PRICE_CHANGE',
    change_reason = '봄맞이 할인 이벤트 (2026-03-01 ~ 2026-03-31)'
WHERE product_id = 1;
```

```sql
SELECT product_id, name, price, change_type, change_reason
FROM product
WHERE product_id = 1;
```

[실행 결과]

product_id	name	price	change_type	change_reason
1	스마트폰 케이스	12000	PRICE_CHANGE	봄맞이 할인 이벤트 (2026-03-01 ~ 2026-03-31)

이제 왜 가격이 변경되었는지 알 수 있다.

재고 조정

재고 실사 후 재고를 조정해보자.

```sql
UPDATE product
SET stock_quantity = 95,
    updated_by = 'warehouse_kim',
    change_type = 'STOCK_ADJUST',
    change_reason = '월말 재고 실사 - 5개 파손 확인'
WHERE product_id = 1;
```

```
SELECT product_id, name, stock_quantity, change_type, change_reason
FROM product
WHERE product_id = 1;
```

[실행 결과]

product_id	name	stock_quantity	change_type	change_reason
1	스마트폰 케이스	95	STOCK_ADJUST	월말 재고 실사 - 5개 파손 확인

상품 판매 중지

상품 판매를 중지해보자.

```
UPDATE product
SET status = 'INACTIVE',
    updated_by = 'admin_lee',
    change_type = 'STATUS_CHANGE',
    change_reason = '제조사 단종 통보'
WHERE product_id = 2;
```

```
SELECT product_id, name, status, change_type, change_reason
FROM product;
```

[실행 결과]

product_id	name	status	change_type	change_reason
1	스마트폰 케이스	ACTIVE	STOCK_ADJUST	월말 재고 실사 - 5개 파손 확인

2	무선 이어폰	INACTIVE	STATUS_CHANGE	제조사 단종 통보

변경 유형 표준화

실무에서는 변경 유형(`change_type`)을 표준화해서 사용하는 것이 좋다. 다음은 예시이다.
참고로 변경 유형은 비즈니스마다 다르기 때문에 담당 부서에서 정해서 관리해야 한다.

change_type	설명
CREATE	신규 등록
PRICE_CHANGE	가격 변경
STOCK_ADJUST	재고 조정
STATUS_CHANGE	상태 변경
INFO_UPDATE	정보 수정 (이름, 설명 등)
PROMOTION	프로모션 적용
CORRECTION	오류 정정

change_type을 사용하는 이유

단순히 `change_reason`에 모든 내용을 적으면 될 것 같은데, 굳이 `change_type`을 따로 나누는 이유는 **데이터 활용과 검색의 효율성** 때문이다.

1. 컴퓨터가 이해하기 쉬운 데이터 (구조화된 데이터)

`change_reason`은 사람이 읽기 위해 자유롭게 작성하는 텍스트다. "가격 변경", "할인 적용", "금액 수정" 등 사람마다 다르게 적을 수 있다. 컴퓨터 입장에서는 이것들이 모두 같은 '가격 변경' 작업인지 파악하기 어렵다. 반면 `change_type`은 `PRICE_CHANGE`처럼 약속된 코드를 사용하므로 시스템이 명확하게 구분할 수 있다.

2. 통계와 검색의 편리함

만약 현업 부서에서 "지난달에 가격이 변경된 상품만 모두 뽑아주세요"라는 요청이 들어왔다고 가정해보자.

`change_type`이 없을 때 (비효율적)

```sql
SELECT * FROM product
WHERE change_reason LIKE '%재고%'
   OR change_reason LIKE '%할인%'
   OR change_reason LIKE '%단종%';
```

검색 조건이 복잡해지고, 오타가 있거나 다른 단어를 쓴 경우 데이터가 누락될 위험이 크다.

`change_type`이 있을 때 (효율적)

```sql
SELECT * FROM product
WHERE change_type = 'PRICE_CHANGE';
```

쿼리가 매우 단순해지고 결과가 정확하다.

따라서 `change_reason`은 **사람을 위한 상세 설명**으로, `change_type`은 **시스템을 위한 분류 코드**로 사용하는 것이 실무적인 설계 방법이다.

장점

이 방식은 다음과 같은 장점이 있다.

1. **변경 이유를 알 수 있다**: 왜 변경되었는지 기록이 남는다.
2. **책임 추적이 가능하다**: 누가, 왜 변경했는지 명확하다.
3. **분쟁 해결에 도움이 된다**: "이 가격은 할인 이벤트 때문입니다"라고 근거를 제시할 수 있다.

남은 문제

- 질문 1: 누가 변경했나요? ➜ 해결
- 질문 2: 언제 변경되었나요? ➜ 해결
- **질문 3: 왜 변경되었나요? ➜ 해결**
- 질문 4: 이전 값은 얼마였나요? ➜ 미해결

이제 질문 4에만 답하면 된다.
질문 4를 알아보기 전에 실무에서 자주 사용하는 더 상세한 감사(Audit) 정보를 기록하는 방법을 알아보자.

변경 추적 컬럼 - 감사(Audit) 컬럼

최근 시스템은 단순히 "누가, 언제, 왜" 변경했는지뿐만 아니라 더 상세한 정보가 필요한 경우가 많다. 특히 다음과 같은 질문에 답해야 할 때가 있다.

- "이 변경이 어떤 시스템에서 발생했나요?" (API 서버? 배치 서버? 관리자 화면?)
- "어떤 IP에서 접속해서 변경했나요?"

감사 컬럼이 필요한 이유

현대 시스템은 하나의 데이터를 여러 경로에서 변경할 수 있다.

1. **웹 관리자 화면**: 운영자가 직접 수정
2. **API 서버**: 외부 연동을 통한 수정
3. **배치 시스템**: 정기 배치 작업에 의한 수정
4. **모바일 앱**: 모바일에서의 수정

문제가 발생했을 때 "어디서 변경되었는지"를 알아야 원인을 쉽고 빠르게 추적할 수 있다.

감사 컬럼 추가

테이블에 감사 컬럼을 추가하자.

```
DROP TABLE IF EXISTS product;

CREATE TABLE product (
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(200) NOT NULL,
    price INT NOT NULL,
    stock_quantity INT NOT NULL DEFAULT 0,
    status VARCHAR(20) NOT NULL DEFAULT 'ACTIVE',
    -- 기본 변경 추적 컬럼
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    created_by VARCHAR(100) NOT NULL,
    updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
```

```
CURRENT_TIMESTAMP,
    updated_by VARCHAR(100) NOT NULL,
    -- 변경 사유 컬럼
    change_type VARCHAR(50),
    change_reason VARCHAR(500),
    -- 감사(Audit) 컬럼
    source_system VARCHAR(50),
    client_ip VARCHAR(50)
);
```

추가된 감사 컬럼의 의미는 다음과 같다.

- `source_system`: 변경이 발생한 시스템 (예: WEB_ADMIN, API, BATCH, MOBILE 등)
- `client_ip`: 요청이 발생한 클라이언트 IP 주소

데이터 등록 - 웹 관리자 화면

관리자가 웹 화면에서 상품을 등록하는 상황이다.

```
INSERT INTO product (
    name, price, stock_quantity, status,
    created_by, updated_by,
    change_type, change_reason,
    source_system, client_ip
) VALUES (
    '스마트폰 케이스', 15000, 100, 'ACTIVE',
    'admin_kim', 'admin_kim',
    'CREATE', '신규 상품 등록',
    'WEB_ADMIN', '192.168.1.3'
);
```

```
SELECT product_id, name, price, source_system, client_ip
FROM product;
```

[실행 결과]

product_id	name	price	source_system	client_ip

1	스마트폰 케이스	15000	WEB_ADMIN	192.168.1.3

- `source_system` 컬럼에 `WEB_ADMIN` 값으로 등록된 것을 확인할 수 있다.
- `client_ip`를 통해 어떤 IP를 통해 들어온 요청인지 확인할 수 있다.

데이터 수정 - API 연동

외부 시스템에서 API를 통해 가격을 변경하는 상황이다.

```
UPDATE product
SET price = 13000,
    updated_by = 'partner_api',
    change_type = 'PRICE_CHANGE',
    change_reason = '외부 프로모션 가격 동기화',
    source_system = 'API',
    client_ip = '123.123.123.123'
WHERE product_id = 1;
```

```
SELECT product_id, name, price, source_system, client_ip, change_reason
FROM product
WHERE product_id = 1;
```

[실행 결과]

product_id	name	price	source_system	client_ip	change_reason
1	스마트폰 케이스	13000	API	123.123.123.123	외부 프로모션 가격 동기화

- `source_system` 컬럼에 `API` 값으로 등록된 것을 확인할 수 있다.
- `client_ip`를 통해 어떤 IP를 통해 들어온 요청인지 확인할 수 있다.

데이터 수정 - 배치 작업

야간 배치 작업에서 재고를 조정하는 상황이다.

```
UPDATE product
SET stock_quantity = 85,
    updated_by = 'batch_system',
    change_type = 'STOCK_ADJUST',
    change_reason = '야간 재고 동기화 배치',
    source_system = 'BATCH',
    client_ip = '10.0.0.5'
WHERE product_id = 1;
```

```
SELECT product_id, name, stock_quantity, source_system, client_ip
FROM product
WHERE product_id = 1;
```

[실행 결과]

product_id	name	stock_quantity	source_system	client_ip
1	스마트폰 케이스	85	BATCH	10.0.0.5

source_system 표준화

실무에서는 시스템 구분을 표준화해서 사용한다. 다음은 예시이다.

source_system	설명
WEB_ADMIN	웹 관리자 화면
WEB_USER	웹 사용자 화면
MOBILE_APP	모바일 앱
API	외부 API 연동

BATCH	배치 시스템
MIGRATION	데이터 마이그레이션
MANUAL	수동 DB 작업 (DBA 직접 수정)

장점

감사 컬럼을 추가하면 다음과 같은 장점이 있다.

1. **문제 발생 시스템 특정**: "이 문제는 배치에서 발생했습니다"라고 명확히 알 수 있다.
2. **보안 감사 대응**: 어떤 IP에서 접근했는지 기록이 남는다.

남은 문제

- **질문 1: 누가 변경했나요? ➜ 해결**
- **질문 2: 언제 변경되었나요? ➜ 해결**
- **질문 3: 왜 변경되었나요? ➜ 해결(강화)**
- 질문 4: 이전 값은 얼마였나요? ➜ 미해결

이번에 학습한 감사 컬럼은 질문 1,2,3을 더 자세히 강화하는 방법이었다.

과거 데이터

이제 이전 값은 얼마인지, 과거 데이터를 확인해야 하는 질문4가 남았다.
질문 4는 사실 다음 3가지 문제를 해결해야 한다.

1. **마지막 변경 정보만 남는다**: 이전 변경 이력은 덮어씌워진다.
2. **이전 값을 알 수 없다**: 변경 전 가격, 변경 전 재고 등을 알 수 없다.
3. **변경 횟수를 알 수 없다**: 몇 번 변경되었는지 파악할 수 없다.

지금까지 배운 방식들은 모두 "현재 테이블에 컬럼을 추가하는 방식"이다. 이 방식으로는 과거 데이터를 보존할 수 없다.
다음 수업에서는 과거 데이터를 보존하는 방법을 알아보겠다.
그 전에 지금까지 학습한 변경 추적 컬럼을 간단히 정리해보자.

> **추적용 요청 ID (correlation ID)**
> 마이크로서비스 아키텍처를 사용한다면 추적용 요청 ID(`request_id`) 추가를 고려하자.

> 하나의 사용자 요청이 여러 서비스를 거칠 때, 동일한 `request_id`를 사용하면 전체 흐름을 쉽게 추적할 수 있다.
> 예를 들어, 고객이 "주문은 했는데 재고가 안 빠졌어요"라고 문의가 들어온다면, 주문 시스템의 로그에서 `request_id`를 찾으면, 동일한 ID로 상품 시스템, 결제 시스템, 배송 시스템의 로그를 모두 찾을 수 있다.

정리

변경 추적 컬럼 정리

지금까지 배운 변경 추적 컬럼들을 정리해보자.

기본 변경 추적 컬럼

가장 기본적인 4개의 컬럼이다.

컬럼명	타입	설명
created_at	DATETIME	최초 등록 시간
created_by	VARCHAR(100)	최초 등록자
updated_at	DATETIME	마지막 수정 시간
updated_by	VARCHAR(100)	마지막 수정자

사용 상황: 거의 모든 테이블에 기본으로 추가하는 것이 좋다. "최소한의 추적"이 필요한 모든 곳에 사용한다.

변경 사유 컬럼

변경 이유를 기록하는 컬럼이다.

컬럼명	타입	설명
change_type	VARCHAR(50)	변경 유형 (PRICE_CHANGE, STATUS_CHANGE 등)
change_reason	VARCHAR(500)	변경 사유 (자유 텍스트)

사용 상황: 변경 이유가 중요한 데이터에 추가한다. 가격, 상태, 등급 등 비즈니스적으로 의미 있는 변경에 사용한다.

감사(Audit) 컬럼

상세한 감사 추적을 위한 컬럼이다.

컬럼명	타입	설명
source_system	VARCHAR(50)	변경 발생 시스템
client_ip	VARCHAR(50)	클라이언트 IP

사용 상황: 여러 시스템에서 데이터를 변경하는 경우, 보안 감사가 필요한 경우에 사용한다.

전체 컬럼 예시

모든 변경 추적 컬럼을 적용한 테이블이다.

```
CREATE TABLE product (
    -- 비즈니스 컬럼
    product_id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(200) NOT NULL,
    price INT NOT NULL,
    stock_quantity INT NOT NULL DEFAULT 0,
    status VARCHAR(20) NOT NULL DEFAULT 'ACTIVE',

    -- 기본 변경 추적 컬럼
    created_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
    created_by VARCHAR(100) NOT NULL,
    updated_at DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    updated_by VARCHAR(100) NOT NULL,

    -- 변경 사유 컬럼
    change_type VARCHAR(50),
    change_reason VARCHAR(500),

    -- 감사(Audit) 컬럼
    source_system VARCHAR(50),
    client_ip VARCHAR(50)
);
```

한계점 정리

변경 추적 컬럼만으로는 해결할 수 없는 문제들이 있다.

질문	답변 가능 여부
누가 마지막으로 수정했나요?	O
언제 마지막으로 수정했나요?	O
왜 수정했나요?	O (change_reason이 있는 경우)
어떤 시스템에서 수정했나요?	O (source_system이 있는 경우)
이전 값은 무엇이었나요?	X
특정 시점의 값은 무엇이었나요?	X
총 몇 번 수정되었나요?	X
모든 변경 이력을 보여주세요	X

결국 "이전 값"과 "변경 이력"을 알기 위해서는 다른 방법이 필요하다. 다음 수업에서는 이력 테이블을 활용한 방법을 알아보겠다.

내용 정리

데이터 변경 이력 설계가 필요한 이유

- 데이터베이스 설계 시 '데이터 변경 이력'은 필수적인 요소다.
- 서비스 운영 중 "과거 가격 확인", "주소 변경 시점", "수정 주체" 등에 대한 질문에 답할 수 있어야 한다.
- 이력이 없으면 고객 클레임 처리, 감사 대응, 데이터 오류 원인 추적이 불가능하다.
- 가격 분쟁, 배송 오류, 금융 감사 등 실무에서 이력 부재로 인한 심각한 문제가 빈번히 발생한다.

이력이 없는 원본 테이블 ~ 문제점 분석

- 기본 테이블 구조에서 `UPDATE`를 수행하면 데이터가 덮어씌워져 과거 정보가 즉시 사라진다.
- "누가, 언제, 왜, 이전 값은 무엇인지"에 대한 질문에 전혀 답할 수 없다.
- 고객이 과거 시점의 데이터(예: 주문 당시 가격)를 근거로 문의할 때 확인 불가능한 심각한 운영 리스크가 있다.

변경 추적 컬럼 - 기본

- 테이블에 `created_at`(등록일), `created_by`(등록자), `updated_at`(수정일), `updated_by`(수정자) 컬럼을 추가한다.
- 데이터의 등록 및 수정 시점에 해당 정보를 함께 기록하여 "누가, 언제" 변경했는지 추적한다.
- 데이터 등록 시에도 `updated_at`에 값을 채워넣는 것이 조회 및 정렬 로직 단순화에 유리하다.
- 이 방식은 이력이 중요하지 않은 마스터 데이터나 단순 내부 관리용 데이터에 적합하다.
- "왜" 변경되었는지와 "이전 값"이 무엇인지는 여전히 알 수 없다.

변경 추적 컬럼 - 변경 사유

- "누가, 언제"를 넘어 "왜" 변경했는지 기록하기 위해 `change_type`(변경 유형)과 `change_reason`(변경 사유) 컬럼을 추가한다.
- `change_type`은 시스템 처리와 검색 효율을 위해 코드로 표준화(예: `PRICE_CHANGE`, `STOCK_ADJUST`)하여 관리한다.
- `change_reason`은 사람이 읽기 위한 구체적인 사유(예: 이벤트명, 단종 사유)를 텍스트로 기록한다.
- 변경 이유와 책임 추적이 가능해져 분쟁 해결에 도움이 되지만, 여전히 이전 값은 알 수 없다.

변경 추적 컬럼 - 감사(Audit) 컬럼

- 변경이 발생한 경로와 환경을 추적하기 위해 `source_system`(시스템 구분), `client_ip`(요청 IP) 컬럼을 추가한다.
- `source_system`은 웹 관리자, API, 배치, 모바일 등 변경 요청의 출처를 구분하여 문제 원인을 빠르게 특정하는 데 사용한다.
- 보안 감사 및 시스템별 오류 추적에 유용하다.
- 마이크로서비스 환경에서는 전체 흐름 추적을 위해 `request_id` 추가를 고려한다.