10. 데이터 무결성

#0.강의/2.데이터베이스로드맵/2.기본

데이터 무결성이 중요한 이유

지금까지 우리는 데이터를 빠르고 효과적으로 조회하고(`SELECT`), 분석하고(`JOIN`, `CASE`), 성능을 높이는(`INDEX`) 방법에 집중해 왔다. 하지만 데이터베이스의 가장 근본적이고 중요한 역할은 데이터를 '안전하게 지키고 관리하는 것'이다.

어느 날, 당신이 쇼핑몰의 월별 매출 보고서를 뽑았는데, 총매출액이 마이너스(-)로 나왔다고 상상해 보자. 황급히 원본 데이터를 살펴보니, 누군가의 실수 혹은 프로그램 버그로 인해 상품의 가격(`price`)에 음수 값이 들어가 있거나, 주문 수량(`quantity`)에 -1이 입력된 것을 발견했다.

또 다른 예로, 분명히 우리 쇼핑몰의 전체 고객은 5명인데, `orders` 테이블에는 존재하지도 않는 `user_id` 99번 고객의 주문 기록이 남아있다.

이런 말도 안 되는 데이터, 현실 세계에서는 결코 존재할 수 없는 데이터를 우리는 **'쓰레기 데이터(Garbage Data)'**라고 부른다.

컴퓨터 과학에는 아주 유명한 격언이 있다.
"Garbage In, Garbage Out (GIGO)"
쓰레기가 들어가면, 쓰레기가 나온다는 뜻이다. 잘못된 데이터가 데이터베이스에 저장되는 순간, 그 데이터를 기반으로 한 모든 분석, 보고서, 그리고 애플리케이션의 동작은 신뢰를 잃고 심각한 오류를 야기하게 된다.

쓰레기 데이터가 초래하는 재앙

- **잘못된 비즈니스 결정**: 매출액이 음수로 찍힌 보고서를 믿고 다음 달 사업 계획을 세울 수는 없다.
- **치명적인 시스템 오류**: 애플리케이션 코드는 상품 가격이 항상 양수일 것이라고 가정하고 만들어졌는데, 음수 가격을 마주치는 순간 예외를 발생시키며 멈춰버릴 수 있다.

- **데이터 불일치**: `users` 테이블에서는 탈퇴한 고객을 삭제했는데, `orders` 테이블에는 그 고객의 주문 기록이 그대로 남아 '주인 없는 주문'이라는 유령 데이터가 되어 떠돌아다닌다.

그렇다면 이 쓰레기 데이터가 애초에 데이터베이스에 저장되지 못하도록 막을 책임은 누구에게 있을까? 물론 데이터를 입력하는 애플리케이션(웹사이트, 앱 등)에서 1차적으로 검증해야 한다. 하지만 그것만으로는 충분하지 않다.

- 애플리케이션에 미처 발견하지 못한 버그가 있을 수 있다.
- 여러 다른 종류의 애플리케이션이 하나의 데이터베이스에 접근할 수도 있다.
- 개발자나 관리자가 데이터베이스에 직접 접속해서 데이터를 수정할 수도 있다.

따라서 우리는 데이터베이스를 **데이터를 지키는 최후의 보루**로 만들어야 한다. 어떤 경로로 데이터 변경 요청이 들어오든, 데이터베이스 스스로가 말도 안 되는 값은 거부하고 데이터의 정확성과 일관성을 지킬 수 있도록 규칙을 설정해야 한다.

이렇게 **데이터의 정확성, 일관성, 유효성을 유지하려는 성질**을 **데이터 무결성(Data Integrity)**이라고 부른다.
그리고 이 데이터 무결성을 강제하기 위해, 테이블의 특정 컬럼에 설정하는 규칙이 바로 **제약 조건(Constraints)**이다.

제약 조건(Constraints)의 역할

제약 조건은 테이블에 데이터를 `INSERT`, `UPDATE`, `DELETE` 할 때, "이 규칙을 어기면 절대 안 돼!" 라고 외치는 문지기와 같다.

- `price` 컬럼에는 0 이상의 값만 받도록 규칙을 정하고, 음수 값을 넣으려는 시도는 거부한다.
- `email` 컬럼에는 중복된 값이 들어올 수 없도록 규칙을 정하고, 이미 가입된 이메일은 거부한다.
- `orders` 테이블의 `user_id`는 반드시 `users` 테이블에 존재하는 `user_id` 값만 받도록 규칙을 정하고, 유령 회원의 주문은 거부한다.

제약 조건을 잘 설정해 둔 데이터베이스는 애플리케이션의 버그나 사용자의 실수로부터 데이터를 지켜내는 튼튼한 성벽이 된다.

우리는 테이블을 생성할 때 `PRIMARY KEY`나 `NOT NULL` 같은 몇 가지 제약 조건을 이미 사용해 보았다. 다음 시간에는 이 제약 조건들의 종류와 각각이 데이터 무결성을 어떻게 지켜주는지, 그 역할에 대해 다시 한번 자세히 파고들어 보겠다.

기본 제약 조건

지난 시간에 우리는 데이터의 정확성과 일관성, 즉 '데이터 무결성'을 지키기 위해 제약 조건이라는 규칙이 왜 필요한지에 대해 배웠다. 제약 조건은 데이터베이스에 '쓰레기 데이터'가 들어오는 것을 막는 최후의 보루 역할을 한다.

우리는 이미 테이블을 처음 만들 때 `NOT NULL`, `UNIQUE`, `PRIMARY KEY` 같은 몇몇 제약 조건들을 사용해왔다. 이번 시간에는 이 제약 조건들이 각각 어떤 종류의 무결성을, 어떻게 지켜주는지 그 역할을 다시 한번 명확히 복습하고 심화하는 시간을 갖겠다.

각 제약 조건이 어떤 잘못된 데이터를 막아주는지, 위반 시에 어떤 에러 메시지를 보여주는지 직접 확인해 보면 그 역할을 확실히 이해할 수 있을 것이다.

`NOT NULL` : NULL 값 방지

- **역할**: 해당 컬럼에 `NULL` 값(값이 없는 상태)이 저장되는 것을 허용하지 않는다. 반드시 필요한 정보가 누락되는 것을 막는다.
- **문법 예시**: `email VARCHAR(255) NOT NULL`

위반 시나리오

회원 가입 시 필수 정보인 이메일을 실수로 `NULL`로 입력하려고 시도해 보자.

```
-- 이메일(NOT NULL)에 NULL 값을 입력 시도 (에러 발생)
INSERT INTO users (name, email) VALUES ('냐옹이', NULL);
```

실행 결과 에러

```
Error Code: 1048. Column 'email' cannot be null
```

데이터베이스는 `email` 컬럼은 `NULL`일 수 없다는 명확한 에러 메시지를 반환하며, 이 `INSERT` 명령을 거부한다. 이렇게 `NOT NULL` 제약 조건은 필수 데이터의 누락을 원천적으로 차단한다.

`UNIQUE` : 중복 값 방지

- **역할**: 해당 컬럼에 들어가는 모든 값은 테이블 내에서 반드시 고유해야(unique) 한다. 중복 데이터가 쌓이는 것을

막는다.

- **문법 예시**: `email VARCHAR(255) UNIQUE`

위반 시나리오

이미 가입된 이메일 주소('sean@example.com')로 다른 사람이 또 가입을 시도하는 상황이다.

```
-- 이메일(UNIQUE)에 중복 값을 입력 시도 (에러 발생)
INSERT INTO users (name, email, address) VALUES ('가짜 션', 'sean@example.com',
'서울시 어딘가');
```

실행 결과 에러

```
Error Code: 1062. Duplicate entry 'sean@example.com' for key 'users.email'
```

데이터베이스는 'sean@example.com' 이라는 값이 `users.email` 키에 대해 중복되었다는 에러와 함께 `INSERT` 를 거부한다.

`PRIMARY KEY` (기본 키): 행의 대표 식별자

- **역할**: 테이블의 각 행을 고유하게 식별할 수 있는 단 하나의 대표 키. `NOT NULL` 과 `UNIQUE` 제약 조건의 특징을 모두 포함한다. 즉, 기본 키 컬럼은 절대 `NULL` 일 수 없으며, 절대 중복될 수 없다.
- **특징**: 테이블당 오직 하나만 설정할 수 있으며, 이 기본 키를 기준으로 데이터가 저장되고 검색되므로 성능에도 매우 중요한 역할을 한다. (MySQL은 기본 키에 자동으로 고성능 인덱스를 생성한다.)
- **문법 예시**: `user_id BIGINT PRIMARY KEY`

위반 시나리오

이미 존재하는 `user_id` 1번으로 새로운 데이터를 삽입하려고 하거나, `id` 를 `NULL` 로 삽입하려고 시도해 보자.

```
-- 기본 키(PRIMARY KEY)에 중복 값을 입력 시도 (에러 발생)
INSERT INTO users (user_id, name, email) VALUES (1, '누군가',
'someone@example.com');
```

실행 결과 에러

```
Error Code: 1062. Duplicate entry '1' for key 'users.PRIMARY'
```

참고로 `users` 테이블의 `user_id`는 `AUTO_INCREMENT`를 사용했다. 따라서 자동 증가하는 값이기 때문에 `NULL`을 입력해도 값이 자동으로 입력된다. 만약 `AUTO_INCREMENT`가 아니라면 PK에 `NULL`을 입력하면 오류가 발생한다.

`DEFAULT` : 기본값 설정

- **역할**: 특정 컬럼에 값을 명시적으로 입력하지 않았을 경우, 자동으로 설정될 기본값을 지정한다. 엄밀히 말해 무결성을 '강제'하는 규칙이라기보다는, 데이터 누락을 방지하고 입력을 편리하게 해주는 기능에 가깝다.
- **문법 예시**: `status VARCHAR(50) DEFAULT 'PENDING'`

주문 테이블 DDL

```sql
CREATE TABLE orders (
    order_id BIGINT AUTO_INCREMENT,
    ...
    status VARCHAR(50) DEFAULT 'PENDING', -- PENDING, COMPLETED, SHIPPED,
CANCELLED
    ...
);
```

활용 시나리오

`orders` 테이블에 주문을 추가할 때, 주문 상태(`status`)를 따로 지정하지 않으면 기본값인 'PENDING'으로 자동 설정된다.

```sql
-- status 컬럼을 생략하고 INSERT
INSERT INTO orders (user_id, product_id, quantity) VALUES (2, 2, 1);
```

이제 방금 넣은 데이터를 조회해 보자.

```sql
SELECT * FROM orders ORDER BY order_id DESC LIMIT 1;
```

[실행 결과]

order_id	user_id	product_id	order_date	quantity	status
8	2	2	...	1	PENDING

- `order_id` 값은 자동 증가하는 값이므로 다른 값이 나타날 수 있다.

`status` 컬럼에 우리가 지정한 기본값인 'PENDING'이 자동으로 들어간 것을 확인할 수 있다.

지금까지 배운 제약 조건들은 모두 **하나의 테이블 내부**에서 데이터의 유효성을 지키는 규칙들이었다. 하지만 데이터베이스의 핵심은 '관계'이며, 이 관계는 여러 테이블에 걸쳐 있다. `orders` 테이블의 `user_id`는 반드시 `users` 테이블에 존재하는 값이어야만 한다.

이처럼 **테이블과 테이블 사이의 관계**의 무결성을 지키는 가장 중요하고 강력한 제약 조건이 아직 남아있다. 다음 시간에는 바로 **외래 키(Foreign Key)** 제약 조건에 대해 심도 있게 알아보겠다.

외래 키 제약 조건

지난 시간에 배운 `NOT NULL`, `UNIQUE`, `PRIMARY KEY`는 모두 하나의 테이블 안에서 데이터의 유효성을 지키는 규칙들이었다. 하지만 데이터베이스의 진정한 힘은 '관계'에 있고, 이 관계는 여러 테이블에 걸쳐서 맺어진다.

여기서 가장 중요한 무결성 원칙, **참조 무결성(Referential Integrity)**이 등장한다. 참조 무결성이란, **두 테이블의 관계가 항상 유효하고 일관된 상태를 유지해야 한다**는 원칙이다.

예를 들어, `orders` 테이블의 `user_id` 컬럼은 반드시 `users` 테이블에 실제로 존재하는 `user_id` 값만을 참조해야 한다. 만약 존재하지도 않는 유령 회원의 주문이 있다면, 이 관계는 깨진 것이다.

이러한 테이블 간의 관계 무결성을 강제하는 가장 강력한 제약 조건이 바로 **외래 키(Foreign Key)**다.

외래 키(FK)의 역할: 유령 데이터를 막아라

외래 키 제약 조건은 두 가지 중요한 규칙을 통해 우리의 데이터를 보호한다.

1. **자식 테이블(`orders`)에 `INSERT`, `UPDATE` 할 때**: 부모 테이블(`users`)에 존재하지 않는 `user_id` 값을 자식 테이블(`orders`)의 `user_id` 컬럼에 넣으려는 시도를 막는다. (유령 주문 생성 방지)
2. **부모 테이블(`users`)에서 `DELETE`, `UPDATE` 할 때**: 자식 테이블(`orders`)에서 참조하고 있는 `user_id` 값을 가진 행을 함부로 삭제하거나 변경하지 못하게 막는다. (기존 주문을 유령 주문으로 만드는 것 방지)

위반 시나리오

우리 테이블에는 이미 외래 키가 설정되어 있다. 이 규칙을 직접 위반해 보자.

1. 유령 주문 생성 시도

존재하지 않는 `user_id`인 999번 고객의 주문을 넣어보자.

```
-- 존재하지 않는 user_id(999)로 주문을 생성 시도 (에러 발생)
INSERT INTO orders (user_id, product_id, quantity) VALUES (999, 1, 1);
```

실행 결과 에러

```
Error Code: 1452. Cannot add or update a child row: a foreign key constraint
fails (`my_shop2`.`orders`, CONSTRAINT `fk_orders_users` FOREIGN KEY
(`user_id`) REFERENCES `users` (`user_id`))
```

데이터베이스는 `fk_orders_users` 외래 키 제약 조건이 실패했다는 명확한 에러를 출력하며, 이 `INSERT`를 거부했다.

2. 주문이 있는 고객 삭제 시도

'션' 고객(`user_id`=1)은 이미 주문 기록이 있다.

`orders` 테이블의 '션' 고객(`user_id`=1)의 주문

```
SELECT * FROM orders WHERE user_id = 1;
```

order_id	user_id	product_id	order_date	quantity	status
1	1	1	2025-06-10 10:00:00	1	COMPLETED
2	1	4	2025-06-10 10:05:00	2	COMPLETED

이 고객을 `users` 테이블에서 삭제하려고 시도해 보자.

```
-- 자식 테이블(orders)에서 참조하고 있는 부모 데이터(users)를 삭제 시도 (에러 발생)
DELETE FROM users WHERE user_id = 1;
```

실행 결과 에러

```
Error Code: 1451. Cannot delete or update a parent row: a foreign key
constraint fails (`my_shop2`.`orders`, CONSTRAINT `fk_orders_users` FOREIGN
KEY (`user_id`) REFERENCES `users` (`user_id`))
```

역시나 데이터베이스는 이 고객을 참조하는 주문이 있기 때문에 삭제할 수 없다는 오류를 발생시키며 삭제를 막는다.

만약 이 삭제 쿼리가 성공한다면 `orders` 테이블에는 '션' 고객(`user_id`=1)의 주문이 남아있지만, `users` 테이블의 션은 제거된다. 결과적으로 주문 내역만 있고, 실제 고객은 사라지는 심각한 문제가 발생한다. 그리고 향후 이 주문이 누구의 주문인지 찾을 수 없게 된다.

만약 부모 테이블의 션(`user_id = 1`) 데이터를 삭제하고 싶다면 다음과 같이 자식 테이블의 션 관련 데이터를 먼저 삭제하고, 이후에 부모 테이블을 삭제하는 순서로 진행해야 한다.

```
DELETE FROM orders WHERE user_id = 1; -- 1.자식 테이블의 션(user_id = 1) 관련 데이터
제거
DELETE FROM users WHERE user_id = 1;  -- 2.부모 테이블의 션(user_id = 1) 관련 데이터
제거
```

`ON DELETE` / `ON UPDATE` 옵션

데이터베이스가 부모 데이터의 삭제나 수정을 무조건 막는 것이 기본값(`RESTRICT`)이며 가장 안전한 정책이다. 하지만 때로는 비즈니스 규칙에 따라 다른 정책이 필요할 수 있다. 예를 들어 "회원이 탈퇴하면, 그 회원의 모든 주문 기록도 함께 삭제되어야 한다" 와 같은 경우다.

이럴 때 사용하는 것이 외래 키의 `ON DELETE`와 `ON UPDATE` 옵션이다.

- `RESTRICT` **(기본값)**: 자식 테이블에 참조하는 행이 있으면 부모 테이블의 행을 삭제/수정할 수 없다. (방금 확인한 동작)
- `CASCADE`: 부모 테이블의 행이 삭제/수정되면, 그를 참조하는 자식 테이블의 행도 **함께 자동으로** 삭제/수정된

다.

- `SET NULL` : 부모 테이블의 행이 삭제/수정되면, 자식 테이블의 해당 외래 키 컬럼의 값을 `NULL` 로 설정한다. (단, 이 옵션을 쓰려면 자식 테이블의 외래 키 컬럼이 `NULL` 을 허용해야 한다.)

`ON DELETE CASCADE` 실습

`CASCADE` 옵션의 강력한 힘을 직접 확인해 보자. 기존 `orders` 테이블을 삭제하고, `ON DELETE CASCADE` 옵션을 추가하여 다시 만들어 보겠다.

> **SQL 소스 파일 참고**
> 강의 자료가 PDF 파일이라 복잡한 SQL 코드를 복사할 때 오류가 발생할 수 있다.
> 이 경우, 섹션 1. 강의 소개와 수업 자료 ➜ SQL 소스 파일을 다운로드해서 사용하자.

```sql
-- 실습을 위해 기존 테이블 삭제 후 CASCADE 옵션으로 재생성
DROP TABLE orders;

CREATE TABLE orders (
    order_id BIGINT AUTO_INCREMENT,
    user_id BIGINT NOT NULL,
    product_id BIGINT NOT NULL,
    order_date DATETIME DEFAULT CURRENT_TIMESTAMP,
    quantity INT NOT NULL,
    status VARCHAR(50) DEFAULT 'PENDING',
    PRIMARY KEY (order_id),

    CONSTRAINT fk_orders_users FOREIGN KEY (user_id)
        REFERENCES users(user_id) ON DELETE CASCADE, -- CASCADE 옵션 추가

    CONSTRAINT fk_orders_products FOREIGN KEY (product_id)
        REFERENCES products(product_id)
);

-- 션 회원 다시 등록
INSERT INTO users(user_id, name, email, address, birth_date) VALUES
(1, '션', 'sean@example.com', '서울시 강남구', '1990-01-15');

-- 주문 데이터 다시 입력
INSERT INTO orders(user_id, product_id, quantity, status) VALUES
(1, 1, 1, 'COMPLETED'),
(1, 4, 2, 'COMPLETED'),
```

```
(2, 2, 1, 'SHIPPED');
```

> **앞서 user_id = 1 션 회원을 삭제했다면, 스크립트를 참고해서 먼저 션 회원을 다시 등록하자.**
> 만약 user_id = 1 션 회원을 삭제하지 않았다면 "션 회원 다시 등록" 스크립트는 실행하지 말자.

새로 저장한 주문 데이터 확인

```
SELECT * FROM orders;
```

order_id	user_id	product_id	order_date	quantity	status
1	1	1	2025-06-30 11:12:23	1	COMPLETED
2	1	4	2025-06-30 11:12:23	2	COMPLETED
3	2	2	2025-06-30 11:12:23	1	SHIPPED

- '션' 고객(`user_id`=1)의 주문이 2건 있는 것을 확인할 수 있다. (`order_id:1,2`)

이제 다시, 주문 기록이 있는 '션' 고객(`user_id`=1)을 삭제해 보자.

```
DELETE FROM users WHERE user_id = 1;
```

- 이번에는 아무 에러 없이 쿼리가 성공적으로 실행된다.

과연 `orders` 테이블은 어떻게 변했을까? 전체 주문 기록을 조회해 보자.

```
SELECT * FROM orders;
```

order_id	user_id	product_id	order_date	quantity	status
3	2	2	2025-06-30 11:12:23	1	SHIPPED

- `users` 테이블에서 `user_id` 1번 고객을 삭제하자, 그를 참조하던 `orders` 테이블의 주문 기록들이 연쇄적

으로 함께 삭제된 것을 확인할 수 있다. (`order_id:1,2` 제거)

`CASCADE` 옵션은 매우 편리하지만, 의도치 않은 대량의 데이터 삭제를 유발할 수 있으므로 반드시 비즈니스 로직을 명확히 이해하고 신중에 신중을 기해서 사용해야 한다.

> **CASCADE - 실무 이야기**
> `CASCADE` 옵션은 분명 편리한 기능이지만, 잘못 사용할 경우 예상치 못한 대량의 데이터가 함께 삭제되는 경우가 있다. 특히 관계가 복잡하게 얽혀 있는 경우에는 파급 효과를 예측하기 어렵다. 이런 문제로 실무에서는 `CASCADE` 옵션은 잘 사용하지 않는다. 대신에 **애플리케이션 계층에서 명시적으로 관련된 데이터를 처리하는 방식**이 더 선호된다.

외래 키는 테이블 간의 관계를 정의하는 것을 넘어, 그 관계가 항상 올바른 상태를 유지하도록 강제하는 데이터베이스의 핵심적인 무결성 장치다.

이제 테이블 간의 관계는 외래 키로 지켰다. 그렇다면 하나의 컬럼 내에서, `NOT NULL` 이나 `UNIQUE` 보다 더 복잡한 비즈니스 규칙(예: 가격은 항상 99보다 커야 한다)을 강제할 방법은 없을까?

다음 시간에는 이러한 규칙을 정의하는 `CHECK` 제약 조건에 대해 알아보겠다.

CHECK 제약 조건

지금까지 우리는 데이터의 존재 여부(`NOT NULL`), 유일성(`UNIQUE`), 테이블 간의 관계(`FOREIGN KEY`) 등 데이터의 '구조'와 '관계'에 대한 무결성을 지키는 제약 조건들을 배웠다.

하지만 아직 해결하지 못한 문제가 남아있다. 데이터의 **'내용'** 자체에 대한 규칙은 어떻게 적용할까?
예를 들어,

- 상품의 가격(`price`)이나 재고 수량(`stock_quantity`)은 절대 음수일 수 없다.
- 할인율(`discount_rate`)은 0%에서 100% 사이의 값이어야 한다.

`NOT NULL` 제약 조건은 가격에 `-5000`이라는 값이 들어오는 것을 막지 못한다. `UNIQUE` 제약 조건은 할인율이 `200`이 되는 것을 막지 못한다. 이런 값들은 형식적으로는 유효하지만, 비즈니스 논리상으로는 명백한 '쓰레기 데이터'다.

이처럼 특정 컬럼에 들어갈 수 있는 값의 범위나 조건을 직접 지정하여, 한층 더 강화된 비즈니스 규칙을 적용하고 싶을 때 사용하는 것이 바로 `CHECK` **제약 조건**이다.

`CHECK` 제약 조건의 역할과 문법

`CHECK` 제약 조건은 특정 컬럼에 대해, `INSERT` 또는 `UPDATE`가 일어날 때마다 지정된 조건식이 '참(`true`)'인지를 검사한다. 만약 조건식이 거짓(`false`)이면, 데이터베이스는 해당 데이터의 입력을 거부하고 에러를 발생시킨다.

문법은 테이블을 생성할 때 컬럼 정의 뒤에 `CHECK (조건식)`을 추가해 주면 된다.

이제 우리 `products` 테이블을 더 튼튼하게 만들기 위해, `CHECK` 제약 조건을 추가하여 다시 설계해 보자.

> **SQL 소스 파일 참고**
> 강의 자료가 PDF 파일이라 복잡한 SQL 코드를 복사할 때 오류가 발생할 수 있다.
> 이 경우, 섹션 1. 강의 소개와 수업 자료 ➜ SQL 소스 파일을 다운로드해서 사용하자.

```sql
-- 실습을 위해 기존 테이블들을 삭제한다.
DROP TABLE IF EXISTS orders;
DROP TABLE IF EXISTS products;

-- CHECK 제약 조건을 추가하여 products 테이블 재생성
CREATE TABLE products (
    product_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    category VARCHAR(100),
    price INT NOT NULL CHECK (price >= 0),
    stock_quantity INT NOT NULL CHECK (stock_quantity >= 0),
    discount_rate DECIMAL(5, 2) DEFAULT 0.00 CHECK (discount_rate BETWEEN 0.00
AND 100.00)
);
```

- `price >= 0`: 가격은 0 이상이어야 한다.
- `stock_quantity >= 0`: 재고 수량은 0 이상이어야 한다.
- `discount_rate BETWEEN 0.00 AND 100.00`: 할인율은 0과 100 사이의 값이어야 한다.
  (`(discount_rate >= 0 AND discount_rate <= 100)` 과 동일)

참고로 제약 조건에 이름을 부여하려면 DDL에서 다음과 같은 SQL을 작성하면 된다.

```
CONSTRAINT products_chk_price CHECK (price >= 0)
```

위반 시나리오

이제 이 튼튼해진 `products` 테이블에 잘못된 데이터를 입력하려고 시도하면 어떤 일이 벌어지는지 확인해 보자.

1. 가격에 음수 입력 시도

```
-- 가격(price)에 음수 값을 입력 시도 (에러 발생)
INSERT INTO products (name, category, price, stock_quantity)
VALUES ('오류상품', '전자기기', -5000, 10);
```

실행 결과 에러

```
Error Code: 3819. Check constraint 'products_chk_1' is violated.
```

데이터베이스가 `products_chk_1`이라는 이름의 `CHECK` 제약 조건(가격은 0 이상)이 위반되었다며, 데이터 입력을 단호하게 거부했다.

2. 할인율에 범위를 벗어난 값 입력 시도

```
-- 할인율(discount_rate)에 100을 초과하는 값을 입력 시도 (에러 발생)
INSERT INTO products (name, category, price, stock_quantity, discount_rate)
VALUES ('초특가상품', '패션', 50000, 20, 120.00);
```

실행 결과 에러

```
Error Code: 3819. Check constraint 'products_chk_3' is violated.
```

이번에도 할인율의 범위를 검사하는 `CHECK` 제약 조건이 '문지기' 역할을 훌륭하게 수행했다.

이처럼 `CHECK` 제약 조건은 애플리케이션 레벨에서 놓칠 수 있는 데이터의 유효성 검사를 데이터베이스가 직접 수행하여, 비즈니스 규칙에 어긋나는 데이터가 저장될 가능성을 원천적으로 차단하는 강력한 수단이다.

이것으로 우리는 데이터의 '상태'가 항상 올바르게 유지되도록 강제하는 다양한 제약 조건들(`NOT NULL`, `UNIQUE`, `PRIMARY KEY`, `FOREIGN KEY`, `CHECK`)에 대해 모두 알아보았다. 이 제약 조건들은 데이터베이스의 무결성을 지키는 든든한 갑옷과 같다.

하지만 데이터베이스의 일관성을 지키기 위해서는 데이터의 '상태'뿐만 아니라, 데이터를 변경하는 '행위'에 대한 규칙도 필요하다.

고객이 '주문하기' 버튼을 누르는 순간을 생각해 보자. 우리 시스템은 최소 두 가지 행위를 해야 한다.

1. **행위 1**: `orders` 테이블에 새로운 주문 정보를 기록한다.
2. **행위 2**: `products` 테이블에서 해당 상품의 재고를 줄인다.

이 두 가지 행위는 무슨 일이 있어도 절대로 쪼개져서는 안 된다. 둘 다 성공적으로 끝나거나, 만약 하나라도 실패하면 둘 다 없었던 일로 되돌려야 한다.

이처럼 여러 개의 작업을 **'All or Nothing(전부 아니면 전무)'** 원칙으로 묶어 데이터의 일관성을 보장하는 가장 중요한 개념, **트랜잭션(Transaction)**에 대해 다음 섹션에서 알아보도록 하겠다.

> **[실무 이야기] 데이터 검증, 어디서 하는 게 좋을까?**
>
> - **요즘 대세는 애플리케이션 코드:** 대부분의 비즈니스 로직과 유효성 검사는 애플리케이션에서 직접 처리한다. 훨씬 유연하고 테스트가 쉽기 때문이다.
> - **DB `CHECK` 제약 조건은 신중하게:** 이런 이유로 데이터베이스의 `CHECK` 제약 조건은 잘 사용하지 않는 추세이다.
> - **'최후의 방어선'으로 활용:** `CHECK` 제약 조건을 쓴다면, 간단하지만 절대 값이 바뀌면 안 되는 핵심 데이터에만 '최후의 방어선'으로 사용하는 것이 좋다.

정리

데이터 무결성이 중요한 이유

- 데이터베이스의 가장 중요한 역할은 데이터를 안전하게 지키고 관리하는 것이다.
- '쓰레기 데이터(Garbage Data)'는 잘못된 비즈니스 결정, 시스템 오류, 데이터 불일치 등 심각한 문제를 야기한다. (Garbage In, Garbage Out)
- 데이터베이스는 애플리케이션의 버그나 사용자의 실수로부터 데이터를 보호하는 최후의 보루 역할을 해야 한다.

- 데이터의 정확성, 일관성, 유효성을 유지하려는 성질을 '데이터 무결성(Data Integrity)'이라고 한다.
- 데이터 무결성을 강제하기 위해 테이블의 특정 컬럼에 설정하는 규칙이 '제약 조건(Constraints)'이다.

기본 제약 조건

- `NOT NULL`: 컬럼에 `NULL` 값이 저장되는 것을 허용하지 않아 필수 정보의 누락을 방지한다.
- `UNIQUE`: 해당 컬럼의 모든 값이 테이블 내에서 고유해야 함을 보장하여 중복 데이터가 쌓이는 것을 막는다.
- `PRIMARY KEY`: 테이블의 각 행을 고유하게 식별하는 대표 키로, `NOT NULL`과 `UNIQUE` 제약 조건의 특징을 모두 포함한다. 테이블당 하나만 설정할 수 있다.
- `DEFAULT`: 특정 컬럼에 값을 명시적으로 입력하지 않았을 경우, 자동으로 설정될 기본값을 지정한다.

외래 키 제약 조건

- '참조 무결성'은 두 테이블 사이의 관계가 항상 유효하고 일관된 상태를 유지해야 한다는 원칙이다.
- '외래 키(Foreign Key)'는 이러한 테이블 간의 관계 무결성을 강제하는 가장 강력한 제약 조건이다.
- 외래 키는 자식 테이블에 존재하지 않는 부모 키 값을 입력하는 것을 막고, 자식 테이블에서 참조하고 있는 부모 데이터를 함부로 삭제하거나 수정하지 못하게 한다.
- `ON DELETE` / `ON UPDATE` 옵션을 통해 부모 데이터가 변경될 때 자식 데이터가 어떻게 동작할지 정책을 설정할 수 있다.
  - `RESTRICT` (기본값): 자식 행이 있으면 부모 행의 변경/삭제를 막는다.
  - `CASCADE`: 부모 행이 변경/삭제되면, 이를 참조하는 자식 행도 함께 자동 변경/삭제된다.
  - `SET NULL`: 부모 행이 변경/삭제되면, 자식 행의 해당 외래 키 컬럼 값을 `NULL`로 설정한다.
- `CASCADE` 옵션은 편리하지만 의도치 않은 대량 데이터 변경을 유발할 수 있으므로, 실무에서는 비즈니스 로직을 명확히 이해하고 신중하게 사용해야 한다.

CHECK 제약 조건

- `CHECK` 제약 조건은 컬럼에 들어갈 수 있는 값의 범위나 조건을 직접 지정하여, 데이터의 '내용'에 대한 비즈니스 규칙을 강제한다.
- `INSERT` 또는 `UPDATE` 시, `CHECK` 조건식이 '참(true)'이 아니면 데이터베이스는 해당 작업을 거부하고 에러를 발생시킨다.
- 예를 들어 '가격은 0 이상이어야 한다(`price >= 0`)' 또는 '할인율은 0과 100 사이여야 한다(`discount_rate BETWEEN 0 AND 100`)'와 같은 규칙을 적용할 수 있다.
- `CHECK` 제약 조건은 애플리케이션 레벨에서 놓칠 수 있는 데이터 유효성 검사를 데이터베이스가 직접 수행하여 데이터 무결성을 강화하는 최후의 수단이다.