

# 10강. Database와 MySQL

## Section 2. 생애 최초 Database 조작하기

1. 디스크와 메모리의 차이를 이해하고,  
Database의 필요성을 이해한다.
2. MySQL Database를 SQL과 함께 조작할 수 있다.

## Section 2. 생애 최초 Database 조작하기

3. 스프링 서버를 이용해 Database에 접근하고 데이터를 저장, 조회, 업데이트, 삭제할 수 있다.
4. API의 예외 상황을 알아보고 예외를 처리할 수 있다.

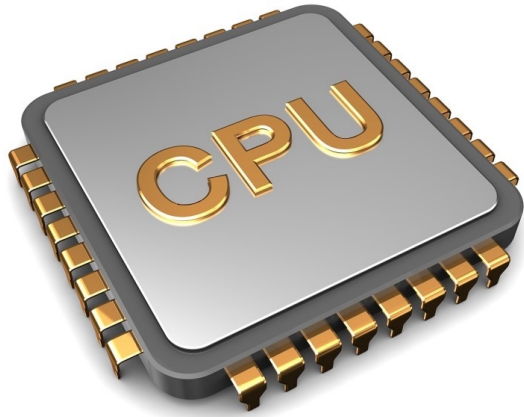
# 지난 시간에...

서버를 종료했다가 시작하면, 유저 정보가 모두 날라갔다!

# 지난 시간에...

마치 저장하지 않은 워드나 한글처럼...

# 컴퓨터의 핵심 부품



CPU

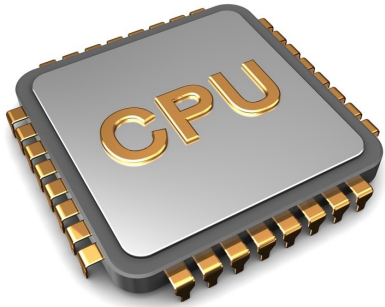


RAM



DISK

# 우리가 서버를 실행시켜 API를 동작시키기까지 일어나는 일



CPU



RAM

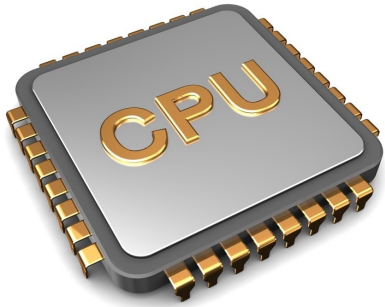


DISK (장기기록)



(1) 개발하고 있는 서버는 DISK에 잠들어 있다.

# 우리가 서버를 실행시켜 API를 동작시키기까지 일어나는 일



CPU



RAM (메모리, 단기기억)



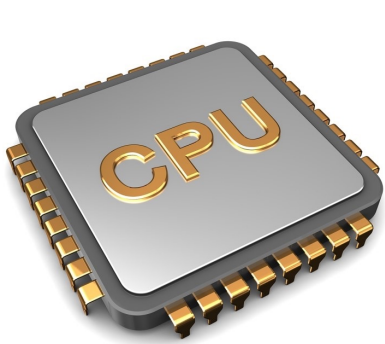
DISK (장기기록)



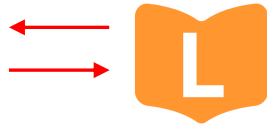
(2) 서버를 실행시키면 DISK에 있는 코드 정보가 RAM으로 복사된다!



# 우리가 서버를 실행시켜 API를 동작시키기까지 일어나는 일



CPU (연산담당)



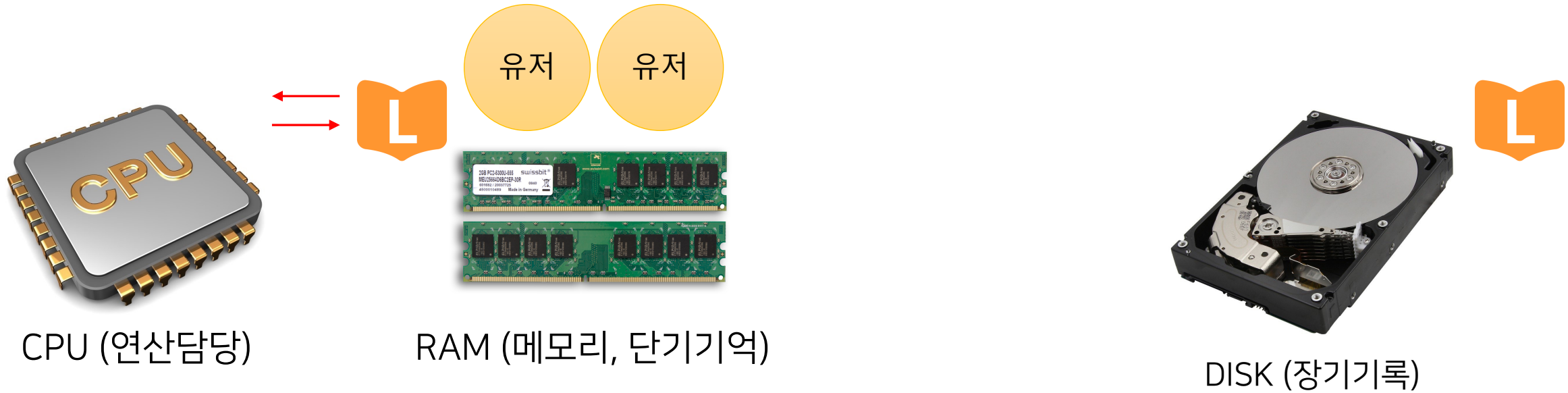
RAM (메모리, 단기기억)



DISK (장기기록)

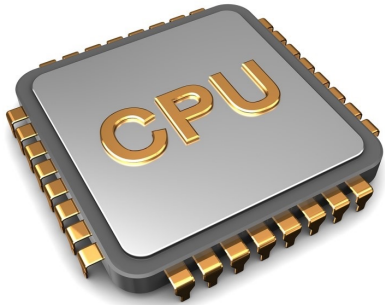
(3) API가 실행되면 '연산'이 수행되며 CPU와 RAM을 왔다갔다 한다

# 우리가 서버를 실행시켜 API를 동작시키기까지 일어나는 일

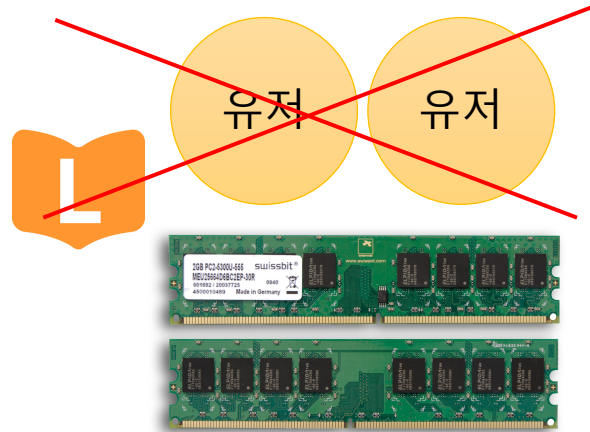


(4) 즉 POST API를 통해 생긴 유저 정보는 RAM에 쓰여 있다!

# 우리가 서버를 실행시켜 API를 동작시키기까지 일어나는 일



CPU (연산담당)



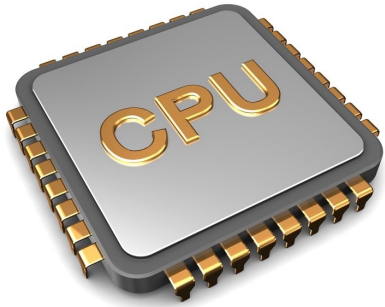
RAM (메모리, 단기기억)



DISK (장기기록)

(5) 서버가 종료되면 RAM에 있는 모든 정보는 사라진다!

# 우리가 서버를 실행시켜 API를 동작시키기까지 일어나는 일



CPU (연산담당)



RAM (메모리, 단기기억)

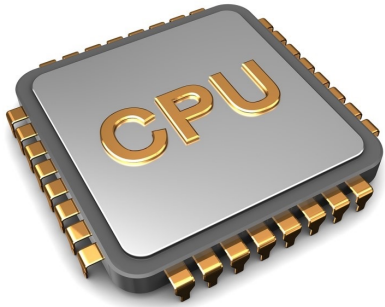


DISK (장기기록)



(6) 때문에 다시 서버를 시작하면! 유저 정보가 없다!

# 우리가 워드, 엑셀에서 저장을 한다는 의미는



CPU (연산담당)



RAM (메모리, 단기 기억)

편집



DISK (장기 기록)



문서에서 편집한 내용을 DISK에 장기기록 한다는 뜻이다!

**그렇다면 서버에서는 어떻게 Disk에 저장할 수 있을까?!**

File 클래스 등을 이용해 직접 Disk에 접근할 수도 있지만~

**그렇다면 서버에서는 어떻게 Disk에 저장할 수 있을까?!**

이럴 때 바로 **Database**를 사용한다!!

# Database란?!

데이터를 **구조화** 시켜 저장하는 친구!



# RDB (Relational Database) - MySQL

데이터를 표처럼 구조화 시켜 저장하는 친구!

# SQL(Structured Query Language)

표처럼 구조화된 데이터를 조회하는 언어

# MySQL에 접근하는 방법!

우선은 MySQL을 가동시켜야 한다!

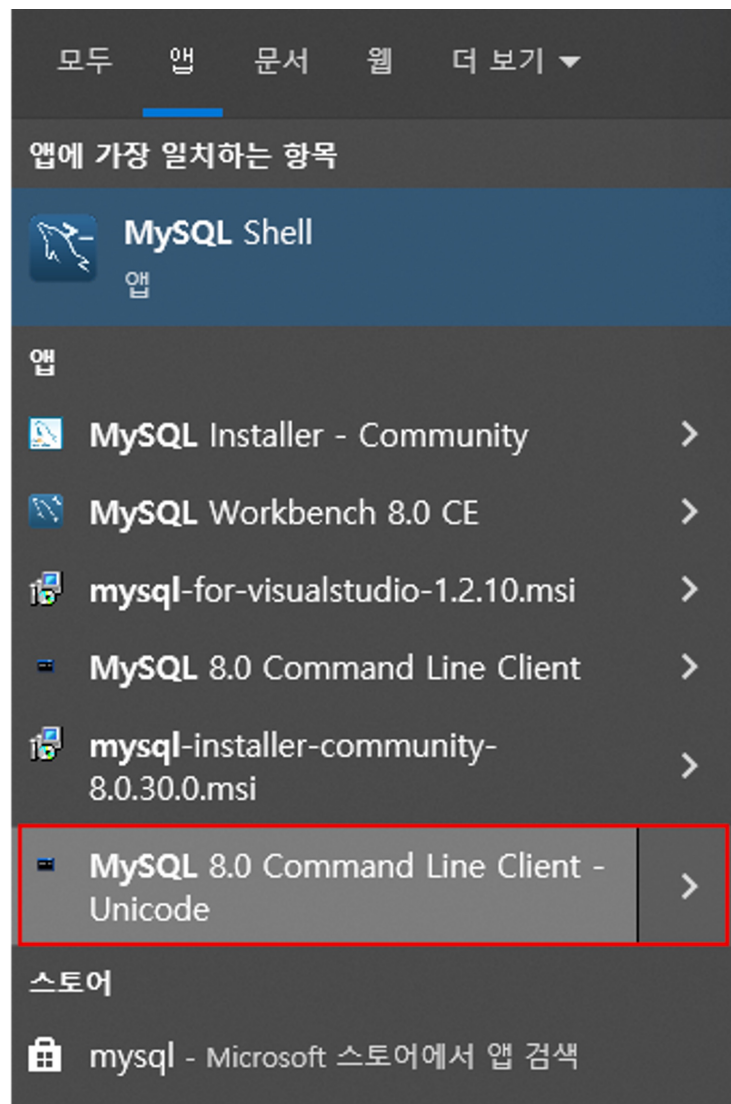
# MySQL에 접근하는 방법 1

IntelliJ Ultimate 사용 가능!

# MySQL에 접근하는 방법 2

IntelliJ를 사용하지 않고 CLI로 접근하기!

# MySQL에 접근하는 방법 2



# MySQL에 접근하는 방법 2



```
mysql -u root -p
```

# 11강. MySQL에서 테이블 만들기



**테이블을 만든다는 의미는 무엇일까?!**

# 컴퓨터에서 엑셀 파일에 과일 정보를 기록한다고 해보자!

1. 우선 엑셀 파일을 담을 폴더를 하나 만든다.
2. 폴더에 들어간다.
3. 폴더 안에 엑셀 파일을 만든다.

# 컴퓨터에서 엑셀 파일에 과일 정보를 기록한다고 해보자!

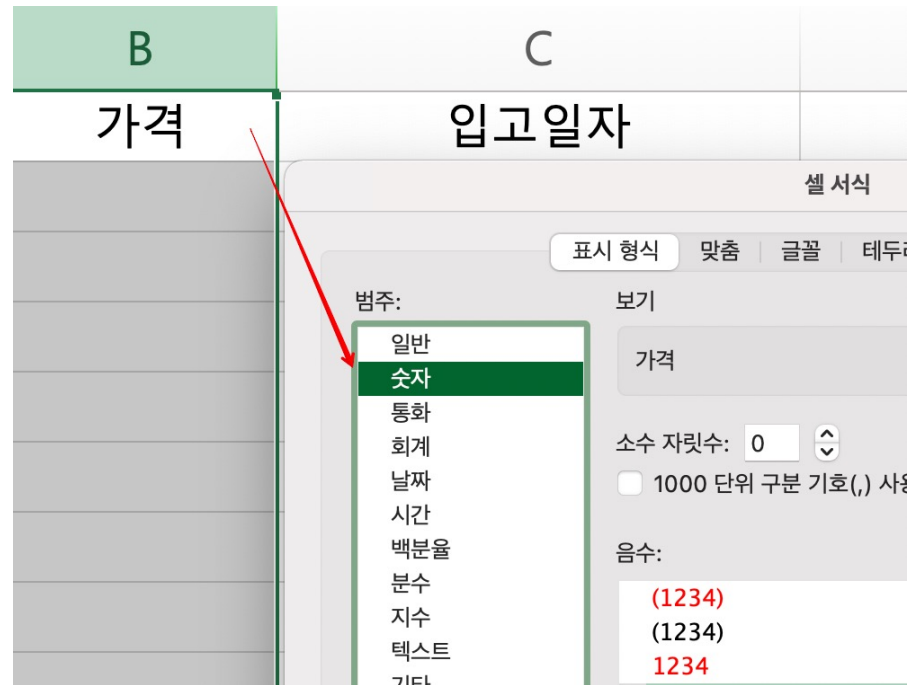
## 4. 엑셀 파일(표)에 Header를 입력한다!

	A	B	C
1	이름	가격	입고일자
2			
3			
4			
5			

# 컴퓨터에서 엑셀 파일에 과일 정보를 기록한다고 해보자!

4. 엑셀 파일(표)에 Header를 입력한다!

5. 각 Header 별로 서식을 지정한다.



# 다시 MySQL로 돌아와서

# 다시 MySQL로 돌아와서

실제 MySQL에 테이블을 만드는 것도 비슷합니다!

# 다시 MySQL로 돌아와서

폴더 = 데이터베이스

# 다시 MySQL로 돌아와서

폴더 = 데이터베이스

엑셀 파일 = 테이블



# 다시 MySQL로 돌아와서

폴더 = 데이터베이스

엑셀 파일 = 테이블

엑셀 파일의 헤더 = 테이블의 필드 정의

# 다시 MySQL로 돌아와서

폴더 = 데이터베이스

엑셀 파일 = 테이블

엑셀 파일의 헤더 = 테이블의 필드 정의

엑셀 파일의 서식 = 테이블의 필드 타입

**MySQL에 접속해서 테이블을 만들어보자!**

# 데이터베이스 만들기

```
create database [데이터베이스 이름];
```

# 데이터베이스 목록 보기

```
show databases;
```

# 데이터베이스 지우기

```
drop database [데이터베이스 이름];
```

# 데이터베이스 안으로 들어가기

```
use [데이터베이스 이름];
```

# 테이블 목록 보기

```
show tables;
```



# 테이블 만들기

```
create table [테이블 이름] (  
    [필드1 이름] [타입] [부가조건],  
    [필드2 이름] [타입] [부가조건],  
    ...  
    primary key ([필드이름])  
);
```

# 테이블 만들기 - 과일 테이블

```
create table fruit
(
    id          bigint auto_increment,
    name        varchar(20),
    price       int,
    stocked_date date,
    primary key (id)
);
```

SQL에 의해 고정된 명령어

# 테이블 만들기 - 과일 테이블

```
create table fruit  
(  
    id          bigint auto_increment,  
    name        varchar(20),  
    price       int,  
    stocked_date date,  
    primary key (id)  
);
```

우리가 만든 테이블 이름

# 테이블 만들기 - 과일 테이블

```
create table fruit
(
  id          bigint auto_increment,
  name        varchar(20),
  price       int,
  stocked_date date,
  primary key (id)
);
```

필드 이름이 나란히 보인다.

# 테이블 만들기 - 과일 테이블

```
create table fruit
(
    id          bigint auto_increment,
    name        varchar(20),
    price       int,
    stocked_date date,
    primary key (id)
);
```

id에 부가 조건을  
auto\_increment라고  
설정해주었다.

# 테이블 만들기 - 과일 테이블

```
create table fruit
(
    id          bigint auto_increment,
    name        varchar(20),
    price       int,
    stocked_date date,
    primary key (id)
);
```

Auto\_increment가  
설정되면,  
데이터를 명시적으로  
넣지 않더라도  
1부터 1씩 증가하며  
자동 기록된다.

# 테이블 만들기 - 과일 테이블

```
create table fruit
(
    id            bigint auto_increment,
    name          varchar(20),
    price         int,
    stocked_date  date,
    primary key (id)
);
```

id라는 필드를  
유일한 키로 지정한다!

사과가 2개 있더라도,  
1번 사과 / 2번 사과가 된다.

# 테이블 만들기 - 과일 테이블

	A	B	C
1	이름	가격	입고일자
2			
3			
4			
5			
6			



# MySQL 타입 살펴보기

# MySQL 타입 살펴보기 - 정수타입

tinyint : 1바이트 정수

int : 4바이트 정수

bigint : 8바이트 정수

# MySQL 타입 살펴보기 - 정수타입

tinyint : 1바이트 정수

int : 4바이트 정수

bigint : 8바이트 정수

id는 혹시나 21억건을 넘을 수도 있으니  
가장 큰 bigint를 사용한다!

# MySQL 타입 살펴보기 - 실수타입

double : 8바이트 정수

decimal(A, B) : 소수점을 B개 가지고 있는 전체 A자릿수 실수

# MySQL 타입 살펴보기 - 실수타입

double : 8바이트 정수

decimal(A, B) : 소수점을 B개 가지고 있는 전체 A자릿수 실수

Decimal(4,2) = 12.23

# MySQL 타입 살펴보기 - 문자열 타입

char(A) : A 글자가 들어갈 수 있는 문자열

varchar(A) : 최대 A 글자가 들어갈 수 있는 문자열

# MySQL 타입 살펴보기 - 날짜, 시간 타입

date : 날짜, yyyy-MM-dd

time : 시간, HH:mm:ss

datetime : 날짜와 시간을 합친 타입, yyyy-MM-dd HH:mm:ss

# 테이블 만들기 - 과일 테이블

```
create table fruit
(
    id            bigint auto_increment,
    name          varchar(20),
    price         int,
    stocked_date  date,
    primary key (id)
);
```



# 테이블 만들기 - 과일 테이블

```
create table fruit
(
    id          bigint auto_increment,
    name        varchar(20),
    price       int,
    stocked_date date,
    primary key (id)
);
```

자동으로 1씩  
올라가는 정수 id

# 테이블 만들기 - 과일 테이블

```
create table fruit
(
    id          bigint auto_increment,
    name        varchar(20),
    price       int,
    stocked_date date,
    primary key (id)
);
```

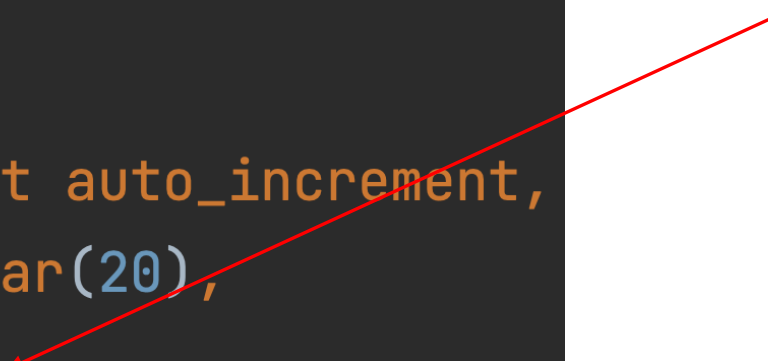
최대 20글자가  
들어가는 과일 이름



# 테이블 만들기 - 과일 테이블

```
create table fruit
(
    id          bigint auto_increment,
    name        varchar(20),
    price       int,
    stocked_date date,
    primary key (id)
);
```

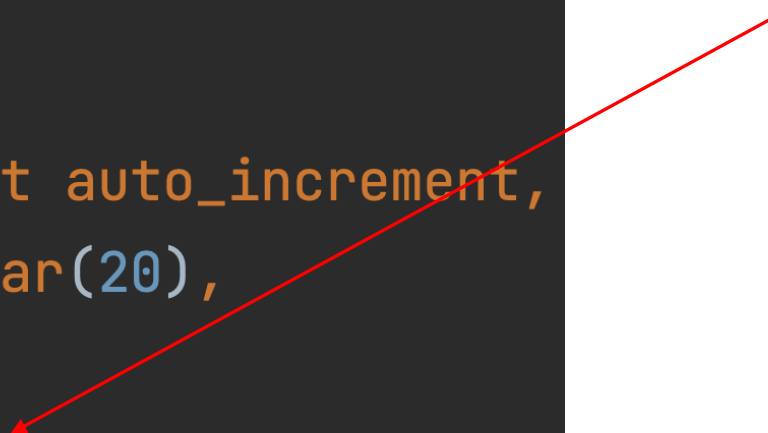
정수가 들어가는  
과일의 가격



# 테이블 만들기 - 과일 테이블

```
create table fruit
(
    id          bigint auto_increment,
    name        varchar(20),
    price       int,
    stocked_date date,
    primary key (id)
);
```

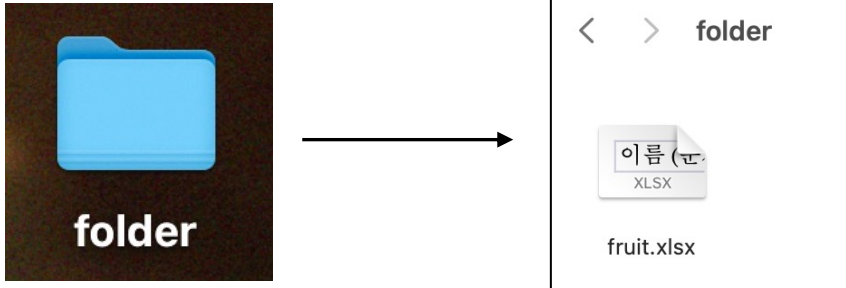
날짜가 들어가는  
재고 일자



# 테이블 만들기 - 과일 테이블

	A	B	C
1	이름 (문자열 - 최대 20자)	가격 (정수)	입고일자 (yyyy-MM-dd)
2			
3			
4			
5			
6			

# 여기까지 요약!



폴더를 만들고 들어가,  
엑셀 파일을 만든다.  
그 후 헤더와 서식을 지정한다.

```
create table fruit
(
    id            bigint auto_increment,
    name          varchar(20),
    price         int,
    stocked_date  date,
    primary key (id)
);
```

데이터베이스를 만들고 들어가,  
테이블을 만든다.  
이때 테이블 이름과 타입, 부가조건을  
지정한다.

# 테이블 제거하기

```
drop table [테이블 이름];
```

# 오늘 배웠던 SQL들은...

DDL(Data Definition Language)이라고 한다.



# 12강. 테이블의 데이터를 조작하기

# fruit 테이블에 과일을 넣고, 조회하고, 수정하고, 삭제하기!

데이터를 넣는다 = 생성, Create

데이터를 조회한다 = 읽기, Retrieve or Read

데이터를 수정한다 = 업데이트, Update

데이터를 삭제한다 = 제거, Delete

# fruit 테이블에 과일을 넣고, 조회하고, 수정하고, 삭제하기!

데이터를 넣는다 = 생성, Create

데이터를 조회한다 = 읽기, Retrieve or Read

데이터를 수정한다 = 업데이트, Update

데이터를 삭제한다 = 제거, Delete

**CRUD**

# 데이터 넣기

```
INSERT INTO [테이블 이름] (필드1이름, 필드2이름, ...)  
VALUES (값1, 값2, ...)
```

# 데이터 넣기

```
INSERT INTO fruit (name, price, stocked_date)  
VALUES ('사과', 1000, '2023-01-01');
```

# 데이터 넣기

```
INSERT INTO fruit (name, price, stocked_date)  
VALUES ('사과', 1000, '2023-01-01');
```

소문자를 사용해도 괜찮다.

# 데이터 넣기

```
INSERT INTO fruit (name, price, stocked_date)  
VALUES ('사과', 1000, '2023-01-01');
```

괄호 안의 필드와 값의 순서가 중요하다.

# 데이터 넣기

```
INSERT INTO fruit (name, price, stocked_date)  
VALUES ('사과', 1000, '2023-01-01');
```

id는 지정해주지 않아도 auto\_increment 덕분에 자동으로 들어간다.



# 데이터 조회하기

```
SELECT * FROM [테이블 이름];
```

# 데이터 조회하기

```
SELECT * FROM fruit;
```

\* 대신 필드 이름을 넣을 수 있다. 여러개도 넣을 수 있다.

# 데이터 조회하기

```
SELECT name, price FROM fruit;
```

# 데이터 조회하기

```
SELECT * FROM fruit;
```

필터를 걸 수 있다!

# 데이터 조회하기

```
SELECT * FROM [테이블 이름] WHERE [조건];
```

# 데이터 조회하기

```
SELECT * FROM fruit WHERE name = '사과';
```

AND 또는 OR을 이용해 조건을 이어 붙일 수 있다!

# 데이터 조회하기

```
SELECT * FROM fruit WHERE name = '사과' AND price <= 2000;
```

조건에는 =, <= 외에도 !=, <, >, >=, between, in, not in 등이 있다.

# 데이터 조회하기

```
SELECT * FROM fruit WHERE price BETWEEN 1000 AND 2000;
```

가격이 1,000원 ~ 2,000인 과일 조회



# 데이터 조회하기

```
SELECT * FROM fruit WHERE name IN ('사과', '수박');
```

이름이 사과 또는 수박인 과일 조회

# 데이터 조회하기

```
SELECT * FROM fruit WHERE name NOT IN ('사과');
```

이름이 사과가 아닌 과일 조회

# 데이터 업데이트하기

```
UPDATE [테이블 이름]  
SET 필드1이름=값, 필드2이름=값, ...  
WHERE [조건];
```

# 데이터 업데이트하기

```
UPDATE fruit SET price=1500 WHERE name = '사과';
```

# 데이터 업데이트하기

주의

만약 [조건]을 붙이지 않으면, 모든 데이터가 업데이트된다!!!

# 데이터 삭제하기

```
DELETE FROM [테이블 이름]  
WHERE [조건];
```

# 데이터 삭제하기

```
DELETE FROM fruit WHERE name = '사과';
```

# 데이터 삭제하기

**주의**

만약 [조건]을 붙이지 않으면, 모든 데이터가 삭제된다!!

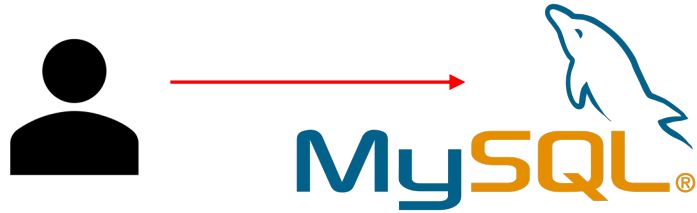


# 오늘 배웠던 SQL들은...

DML(Data Manipulation Language)이라고 한다.

# 13강. Spring에서 Database 사용하기

# 지금까지는 사람이 직접 MySQL에 접근했다!



# 이제 우리의 스프링 서버가 MySQL DB에 접근하게 하자!



# application.yml 만들고 설정하기!

```
spring:
  datasource:
    url: "jdbc:mysql://localhost/library"
    username: "root"
    password: ""
    driver-class-name: com.mysql.cj.jdbc.Driver
```

# application.yml 만들고 설정하기!

```
spring:
  datasource:
    url: "jdbc:mysql://localhost/library"
    username: "root"
    password: ""
    driver-class-name: com.mysql.cj.jdbc.Driver
```

jdbc:mysql:// - jdbc를 이용해 mysql에 접근한다!

# application.yml 만들고 설정하기!

```
spring:
  datasource:
    url: "jdbc:mysql://localhost/library"
    username: "root"
    password: ""
    driver-class-name: com.mysql.cj.jdbc.Driver
```

localhost - 접근하려는 mysql은 localhost에 있다.

# application.yml 만들고 설정하기!

```
spring:
  datasource:
    url: "jdbc:mysql://localhost/library"
    username: "root"
    password: ""
    driver-class-name: com.mysql.cj.jdbc.Driver
```

/library – 접근하려는 DB는 library이다.



# application.yml 만들고 설정하기!

```
spring:
  datasource:
    url: "jdbc:mysql://localhost/library"
    username: "root"
    password: ""
    driver-class-name: com.mysql.cj.jdbc.Driver
```

MySQL에 접근하기 위한 계정명

# application.yml 만들고 설정하기!

```
spring:
  datasource:
    url: "jdbc:mysql://localhost/library"
    username: "root"
    password: ""
    driver-class-name: com.mysql.cj.jdbc.Driver
```

MySQL에 접근하기 위한 비밀번호

# application.yml 만들고 설정하기!

```
spring:
  datasource:
    url: "jdbc:mysql://localhost/library"
    username: "root"
    password: ""
    driver-class-name: com.mysql.cj.jdbc.Driver
```

데이터베이스에 접근할 때 사용할 프로그램

# 본격적으로 코딩을 하기 전에~!!

우리 데이터베이스에 **user** 테이블을 만들자!

# 본격적으로 코딩을 하기 전에~!!

```
create table user (  
    id    bigint auto_increment,  
    name  varchar(25),  
    age   int,  
    primary key (id)  
);
```

# POST API 변경!

```
@RestController
public class UserController {

    2 usages
    private final JdbcTemplate jdbcTemplate;

    public UserController(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @PostMapping("/user")
    public void saveUser(@RequestBody UserCreateRequest request) {
        String sql = "INSERT INTO user(name, age) VALUES(?, ?)";
        jdbcTemplate.update(sql, request.getName(), request.getAge());
    }
}
```

jdbcTemplate을 이용해  
SQL을 날릴 수 있다!

생성자를 만들어  
jdbcTemplate을 파라미터로  
넣으면, 자동으로 들어온다!

# POST API 변경!

```
@RestController
public class UserController {

    2 usages
    private final JdbcTemplate jdbcTemplate;

    public UserController(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @PostMapping("/user")
    public void saveUser(@RequestBody UserCreateRequest request) {
        String sql = "INSERT INTO user(name, age) VALUES(?, ?)";
        jdbcTemplate.update(sql, request.getName(), request.getAge());
    }
}
```

Section3에서  
자세한 동작 원리를  
설명드립니다!

# POST API 변경!

```
@RestController
public class UserController {

    2 usages
    private final JdbcTemplate jdbcTemplate;

    public UserController(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @PostMapping("/user")
    public void saveUser(@RequestBody UserCreateRequest request) {
        String sql = "INSERT INTO user(name, age) VALUES(?, ?)";
        jdbcTemplate.update(sql, request.getName(), request.getAge());
    }
}
```

SQL을 만들어 문자열 변수로 저장합니다.

값이 들어가는 부분에 ?를 사용하면, 값을 유동적으로 넣을 수 있습니다.



# POST API 변경!

```
@RestController
public class UserController {

    2 usages
    private final JdbcTemplate jdbcTemplate;

    public UserController(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @PostMapping("/user")
    public void saveUser(@RequestBody UserCreateRequest request) {
        String sql = "INSERT INTO user(name, age) VALUES(?, ?)";
        jdbcTemplate.update(sql, request.getName(), request.getAge());
    }
}
```

jdbcTemplate.update( )는

INSERT  
UPDATE  
DELETE

쿼리에 사용할 수 있다.

# POST API 변경!

```
@RestController
public class UserController {

    2 usages
    private final JdbcTemplate jdbcTemplate;

    public UserController(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    @PostMapping("/user")
    public void saveUser(@RequestBody UserCreateRequest request) {
        String sql = "INSERT INTO user(name, age) VALUES(?, ?)";
        jdbcTemplate.update(sql, request.getName(), request.getAge());
    }
}
```

첫 파라미터로는 sql을 받고,  
?를 대신할 값을  
차례로 넣으면 된다.

# GET API 변경!

```
@GetMapping("/user")
public List<UserResponse> getUsers() {
    String sql = "SELECT * FROM user";
    return jdbcTemplate.query(sql, new RowMapper<UserResponse>() {
        @Override
        public UserResponse mapRow(ResultSet rs, int rowNum) throws SQLException {
            long id = rs.getLong(columnLabel: "id");
            String name = rs.getString(columnLabel: "name");
            int age = rs.getInt(columnLabel: "age");
            return new UserResponse(id, name, age);
        }
    });
}
```

jdbcTemplate.query(sql, RowMapper 구현 익명클래스)

# GET API 변경!

```
@GetMapping("/user")
public List<UserResponse> getUsers() {
    String sql = "SELECT * FROM user";
    return jdbcTemplate.query(sql, new RowMapper<UserResponse>() {
        @Override
        public UserResponse mapRow(ResultSet rs, int rowNum) throws SQLException {
            long id = rs.getLong(columnLabel: "id");
            String name = rs.getString(columnLabel: "name");
            int age = rs.getInt(columnLabel: "age");
            return new UserResponse(id, name, age);
        }
    });
}
```

RowMapper는 쿼리의 결과를 받아, 객체를 반환한다!

# GET API 변경!

```
@GetMapping("/user")
public List<UserResponse> getUsers() {
    String sql = "SELECT * FROM user";
    return jdbcTemplate.query(sql, new RowMapper<UserResponse>() {
        @Override
        public UserResponse mapRow(ResultSet rs, int rowNum) throws SQLException {
            long id = rs.getLong(columnLabel: "id");
            String name = rs.getString(columnLabel: "name");
            int age = rs.getInt(columnLabel: "age");
            return new UserResponse(id, name, age);
        }
    });
}
```

ResultSet에 getType("필드이름")을 사용해  
실제 값을 가져올 수 있다.

# GET API 변경!

```
@GetMapping("/user")
public List<UserResponse> getUsers() {
    String sql = "SELECT * FROM user";
    return jdbcTemplate.query(sql, new RowMapper<UserResponse>() {
        @Override
        public UserResponse mapRow(ResultSet rs, int rowNum) throws SQLException {
            long id = rs.getLong(columnLabel: "id");
            String name = rs.getString(columnLabel: "name");
            int age = rs.getInt(columnLabel: "age");
            return new UserResponse(id, name, age);
        }
    });
}
```

UserResponse에 생성자를 추가해주었다.

# GET API 변경!

Java 랴다를 사용하면 조금 더 간결하게 변경할 수 있다!

# GET API 변경!

```
@GetMapping(🌐📄"/user")
public List<UserResponse> getUsers() {
    String sql = "SELECT * FROM user";
    return jdbcTemplate.query(sql, (rs, rowNum) -> {
        long id = rs.getLong(columnLabel: "id");
        String name = rs.getString(columnLabel: "name");
        int age = rs.getInt(columnLabel: "age");
        return new UserResponse(id, name, age);
    });
}
```



# 14강. 유저 업데이트 API, 삭제 API 개발과 테스트

# 도서관 사용자 이름을 업데이트 할 수 있다.

- HTTP Method : PUT
- HTTP Path : /user
- HTTP Body (JSON)

```
{  
  "id": Long,  
  "name": String // 변경되어야 하는 이름이 들어온다  
}
```

- 결과 반환 X (HTTP 상태 200 OK이면 충분한다)


# 도서관 사용자를 삭제할 수 있다.

- HTTP Method : DELETE
- HTTP Path : /user
- 쿼리 사용
  - 문자열 name (삭제되어야 하는 사용자 이름)
- 결과 반환 X (HTTP 상태 200 OK이면 충분한다)

# 도서관 사용자 이름을 업데이트 할 수 있다.

```
@PostMapping("/user")
public void updateUser(@RequestBody UserUpdateRequest request) {
    String sql = "UPDATE user SET name = ? WHERE id = ?";
    jdbcTemplate.update(sql, request.getName(), request.getId());
}
```

# 도서관 사용자를 삭제할 수 있다.

```
@DeleteMapping("/user")  
public void deleteUser(@RequestParam String name) {  
    String sql = "DELETE FROM user WHERE name = ?";  
    jdbcTemplate.update(sql, name);  
}
```

**웹 UI를 이용해 확인해보자!**

# 한 가지 생각해볼 내용!

업데이트나 삭제를 할 때, 존재하지 않는 유저라면?!

# 한 가지 생각해볼 내용!

DELETE

Params ☒ Authorization Headers (8) Body ☒ Pre-request Script Tests Settings Cookies

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/>	name	ABC			
	Key	Value	Description		

Body Cookies Headers (4) Test Results

☒ Pretty Raw Preview Visualize Text

☒ Status: 200 OK Time: 7 ms Size: 123 B

존재하지 않는 유저라도 성공했다는 의미인 200 OK를 전달한다!



# 한 가지 생각해볼 내용!


다음 시간에, 이런 예외 상황을 처리해보도록 하자!

# 15강. 유저 업데이트 API, 삭제 API 예외 처리 하기

# 문제 상황을 정리해보자!

없는 유저를 업데이트 하거나 삭제하려 해도 200 OK가 나온다!

# API에서 예외를 던지면 어떻게 될까?

```
@GetMapping( "/user/error-test")  
public void errorTest() {  
    throw new IllegalArgumentException();  
}
```

# POST MAN을 이용해 확인해보자!

http://localhost:8080/user/error-test

Save

GET

http://localhost:8080/user/error-test

Send

Params

Authorization

Headers (6)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

	KEY	VALUE	DESCRIPTION	...	Bulk Ed
	Key	Value	Description		

Body

Cookies

Headers (4)

Test Results

Status: 500 Internal Server Error

Time: 202 ms

Size: 270 B

Save Response

Pretty

Raw

Preview

Visualize

JSON

```
1 {
2   "timestamp": "2022-08-31T12:14:52.347+00:00",
3   "status": 500,
4   "error": "Internal Server Error",
5   "path": "/user/error-test"
6 }
```

# API에서 예외를 던지면 어떻게 될까?

200 OK 대신 **500 Internal Server Error**가 나오게 된다!

# API에서 데이터 존재 여부를 확인해 예외를 던지자!

```
@PostMapping("/user")
public void updateUser(@RequestBody UserUpdateRequest request) {
    String readSql = "SELECT * FROM user WHERE id = ?";
    boolean isUserNotExist = jdbcTemplate.query(readSql, (rs, rowNum) -> 0, request.getId()).isEmpty();
    if (isUserNotExist) {
        throw new IllegalArgumentException();
    }

    String updateSql = "UPDATE user SET name = ? WHERE id = ?";
    jdbcTemplate.update(updateSql, request.getName(), request.getId());
}
```

# API에서 데이터 존재 여부를 확인해 예외를 던지자!

```
@PostMapping("/user")
public void updateUser(@RequestBody UserUpdateRequest request) {
    String readSql = "SELECT * FROM user WHERE id = ?";
    boolean isUserNotExist = jdbcTemplate.query(readSql, (rs, rowNum) -> 0, request.getId()).isEmpty();
    if (isUserNotExist) {
        throw new IllegalArgumentException();
    }

    String updateSql = "UPDATE user SET name = ? WHERE id = ?";
    jdbcTemplate.update(updateSql, request.getName(), request.getId());
}
```

id를 기준으로 유저가 존재하는지  
확인하기 위해 SELECT 쿼리를 작성했다.



# API에서 데이터 존재 여부를 확인해 예외를 던지자!

```
@PostMapping("/user")
public void updateUser(@RequestBody UserUpdateRequest request) {
    String readSql = "SELECT * FROM user WHERE id = ?";
    boolean isUserNotExist = jdbcTemplate.query(readSql, (rs, rowNum) -> 0, request.getId()).isEmpty();
    if (isUserNotExist) {
        throw new IllegalArgumentException();
    }

    String updateSql = "UPDATE user SET name = ? WHERE id = ?";
    jdbcTemplate.update(updateSql, request.getName(), request.getId());
}
```

SQL을 날려 DB에 데이터가 있는지 확인한다!

# API에서 데이터 존재 여부를 확인해 예외를 던지자!



```
jdbcTemplate.query(readSql, (rs, rowNum) -> 0, request.getId())
```

```
SELECT * FROM user WHERE id = request.getId()
```

# API에서 데이터 존재 여부를 확인해 예외를 던지자!

```
jdbcTemplate.query(readSql, (rs, rowNum) -> 0, request.getId())
```

SELECT SQL의 결과가 있으면 0으로 변환된다!

# API에서 데이터 존재 여부를 확인해 예외를 던지자!

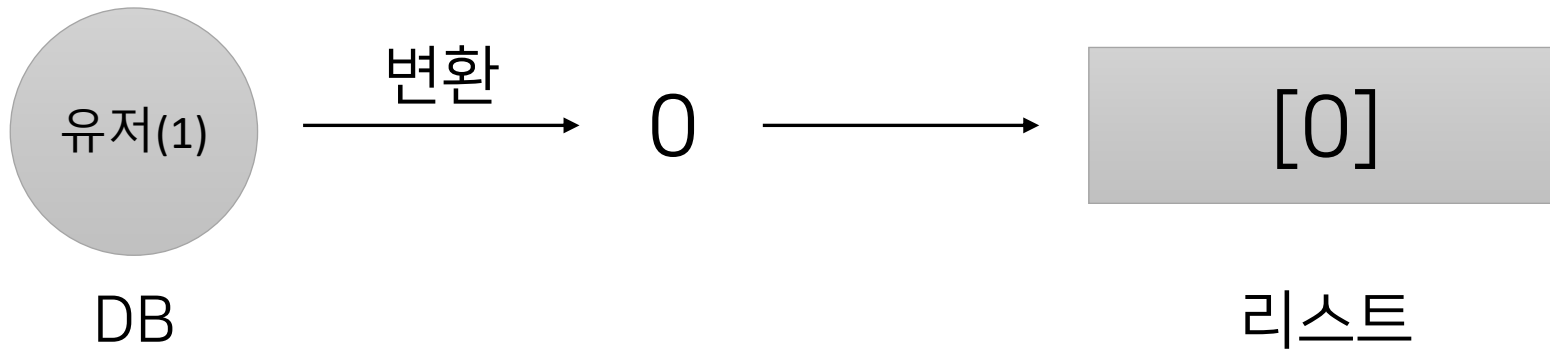
```
jdbcTemplate.query(readSql, (rs, rowNum) -> 0, request.getId())
```

그리고 0은 최종적으로 **List**로 반환된다!

# API에서 데이터 존재 여부를 확인해 예외를 던지자!

```
jdbcTemplate.query(readSql, (rs, rowNum) -> 0, request.getId())
```

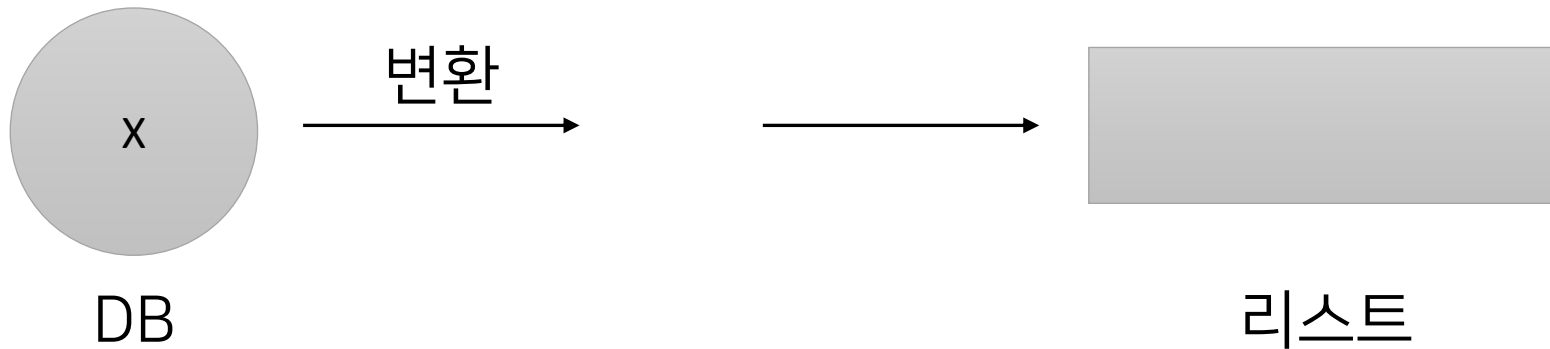
결론적으로 해당 id를 가진 유저가 있으면 **0이 담긴 List**가 나오고



# API에서 데이터 존재 여부를 확인해 예외를 던지자!

```
jdbcTemplate.query(readSql, (rs, rowNum) -> 0, request.getId())
```

결론적으로 해당 id를 가진 유저가 없으면 **빈 List**가 나온다.



# API에서 데이터 존재 여부를 확인해 예외를 던지자!

```
@PostMapping("/user")
public void updateUser(@RequestBody UserUpdateRequest request) {
    String readSql = "SELECT * FROM user WHERE id = ?";
    boolean isUserNotExist = jdbcTemplate.query(readSql, (rs, rowNum) -> 0, request.getId()).isEmpty();
    if (isUserNotExist) {
        throw new IllegalArgumentException();
    }

    String updateSql = "UPDATE user SET name = ? WHERE id = ?";
    jdbcTemplate.update(updateSql, request.getName(), request.getId());
}
```

jdbcTemplate.query()의 결과인  
List가 비어 있다면, 유저가 없다는 뜻이다!

# API에서 데이터 존재 여부를 확인해 예외를 던지자!

```
@PostMapping("/user")
public void updateUser(@RequestBody UserUpdateRequest request) {
    String readSql = "SELECT * FROM user WHERE id = ?";
    boolean isUserNotExist = jdbcTemplate.query(readSql, (rs, rowNum) -> 0, request.getId()).isEmpty();
    if (isUserNotExist) {
        throw new IllegalArgumentException();
    }

    String updateSql = "UPDATE user SET name = ? WHERE id = ?";
    jdbcTemplate.update(updateSql, request.getName(), request.getId());
}
```

만약 사용자가 존재하지 않는다면  
IllegalArgumentException을 던진다.



**이제 POST MAN을 이용해 확인해보자!**

**삭제 API도 비슷하게 적용해보자!**

# 16강. Section2 정리. 다음으로!

# 생애 최초 Database 조작하기

1. 디스크와 메모리 차이를 이해하고, Database의 필요성을 이해한다.
2. SQL을 이용해 MySQL Database를 조작할 수 있다.
3. 스프링 서버를 이용해 Database에 접근하고 데이터를 저장, 조회, 업데이트, 삭제할 수 있다.
4. API의 예외 상황을 알아보고 예외를 처리할 수 있다.

# 하지만...!

현재 한 가지 문제가 존재한다.

# 하지만...!

한 클래스인 Controller에서 너무 많은 역할을 하고 있다.

# 하지만...!

한 API에서 무려 10개의 Table을 사용해야 한다면?!!!

# 다음 시간에 이어서

왜 한 함수에서 모든 기능을 구현하면 안되는지



# 다음 시간에 이어서

이 문제를 스프링을 이용해 어떻게 해결할 수 있는지

**감사합니다**