

# **23강. 문자열 SQL을 직접 사용하는 것이 너무 어렵다!!**

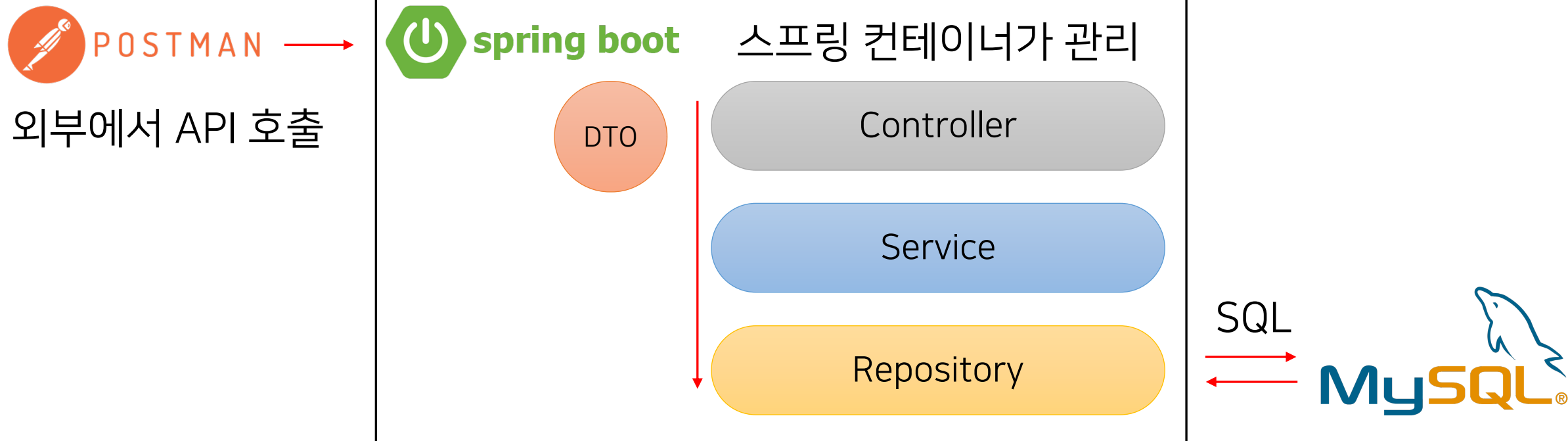
## Section 4. 생애 최초 JPA 사용하기

1. 문자열 SQL을 직접 사용하는 것의 한계를 이해하고,  
해결책인 JPA, Hibernate, Spring Data JPA가 무엇인지 이해한다.
2. Spring Data JPA를 이용해 데이터를 생성, 조회,  
수정, 삭제할 수 있다.

## Section 4. 생애 최초 JPA 사용하기

3. 트랜잭션이 왜 필요한지 이해하고,  
스프링에서 트랜잭션을 제어하는 방법을 익힌다.
4. 영속성 컨텍스트와 트랜잭션의 관계를 이해하고,  
영속성 컨텍스트의 특징을 알아본다.

# 지금까지 우리가 작성한 코드를 살펴보면...



# SQL을 직접 작성해서 사용했다!



# SQL을 직접 작성하면 어떤 점이 아쉬울까?!

1. 문자열을 작성하기 때문에 실수할 수 있고, 실수를 인지하는 시점이 느리다!

```
1 usage
public void saveUser(String name, Integer age) {
    String sql = "INSERT INTO users (name, age) VALUES(?, ?)";
    jdbcTemplate.update(sql, name, age);
}
```

# SQL을 직접 작성하면 어떤 점이 아쉬울까?!

컴파일 시점에 발견되지 않고,  
런타임 시점에 발견된다!

# SQL을 직접 작성하면 어떤 점이 아쉬울까?!

2. 특정 데이터베이스에 종속적이게 된다.





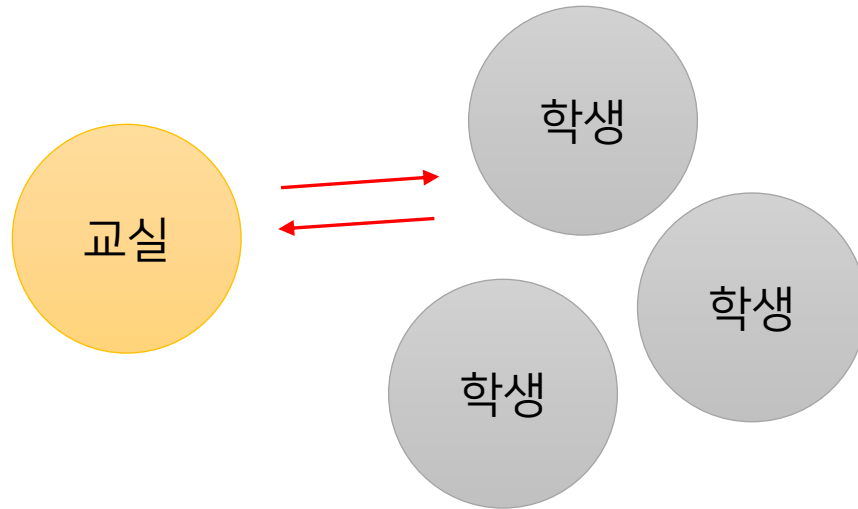
# SQL을 직접 작성하면 어떤 점이 아쉬울까?!

3. 반복 작업이 많아진다. 테이블을 하나 만들 때마다 CRUD 쿼리가 항상 필요하다.

```
1 usage
public List<UserResponse> getUserResponses() {
    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);
    });
}
```

# SQL을 직접 작성하면 어떤 점이 아쉬울까?!

4. 데이터베이스의 테이블과 객체는 패러다임이 다르다.

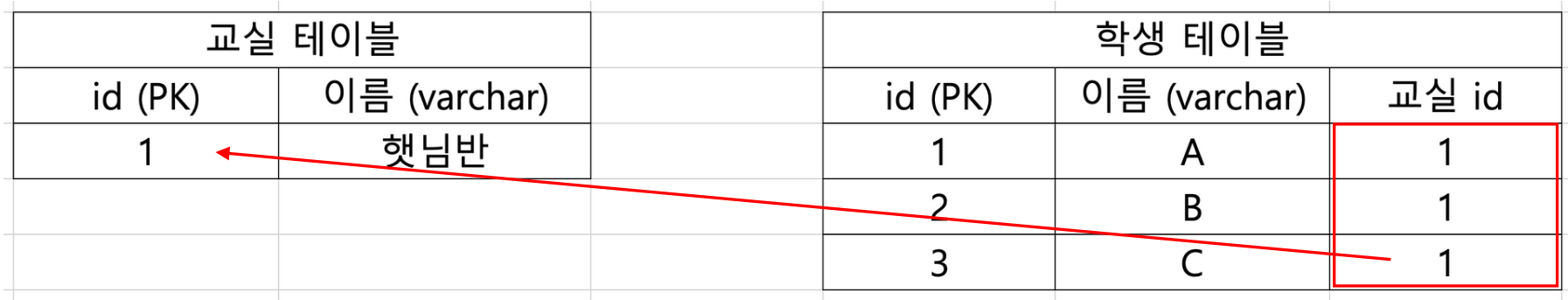


교실과 학생은 서로를 가리킬 수 있다.

# SQL을 직접 작성하면 어떤 점이 아쉬울까?!

4. 데이터베이스의 테이블과 객체는 패러다임이 다르다.

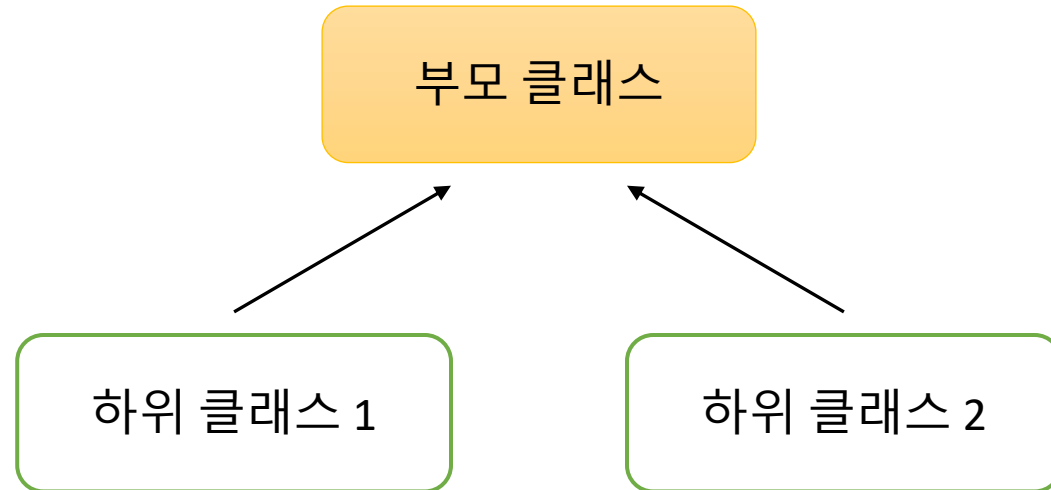
교실 테이블		학생 테이블		
id (PK)	이름 (varchar)	id (PK)	이름 (varchar)	교실 id
1	햇님반	1	A	1
		2	B	1
		3	C	1



학생만 교실을 가리킨다.

# SQL을 직접 작성하면 어떤 점이 아쉬울까?!

4. 데이터베이스의 테이블과 객체는 패러다임이 다르다.



# 아쉬운 점을 정리해보면...

1. 문자열을 작성하기 때문에 실수할 수 있고, 실수를 인지하는 시점이 느리다!
2. 특정 데이터베이스에 종속적이게 된다.
3. 반복 작업이 많아진다. 테이블을 하나 만들 때마다 CRUD 쿼리가 항상 필요하다.
4. 데이터베이스의 테이블과 객체는 패러다임이 다르다.

**그래서 등장했습니다!**

JPA

# 그래서 등장했습니다!

JPA (Java Persistence API)  
자바 진영의 ORM (Object-Relational Mapping)

# 하나씩 살펴봅시다!

JPA (**Java** Persistence API)

자바 진영의 ORM (Object-Relational Mapping)



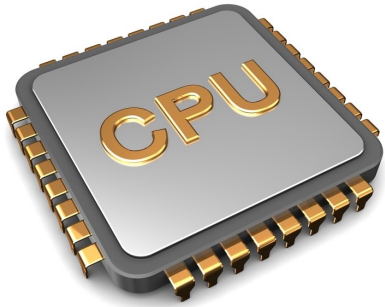
# 하나씩 살펴봅시다!

JPA (Java **Persistence** API)

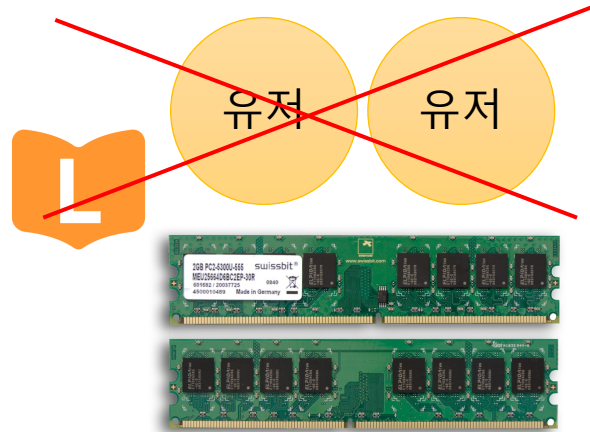
자바 진영의 ORM (Object-Relational Mapping)

영속성

# 우리가 서버를 실행시켜 API를 동작시키기까지 일어나는 일



CPU (연산담당)



RAM (메모리, 단기기억)



DISK (장기기록)

서버가 종료되면 RAM에 있는 모든 정보는 사라진다!

# 하나씩 살펴봅시다!

JPA (Java **Persistence** API)

자바 진영의 ORM (Object-Relational Mapping)

영속성 : 서버가 재시작되어도 데이터는 영구적으로 저장되는 속성

# 하나씩 살펴봅시다!

JPA (Java Persistence **API**)  
자바 진영의 ORM (Object-Relational Mapping)

API : 정해진 규칙

# 하나씩 살펴봅시다!

**JPA** (Java Persistence API)

자바 진영의 ORM (Object-Relational Mapping)

데이터를 영구적으로 보관하기 위해 Java 진영에서 정해진 규칙

# 하나씩 살펴봅시다!

JPA (Java Persistence API)

자바 진영의 ORM (**Object**-Relational Mapping)

```
public class User {  
  
    2 usages  
    private String name;  
    2 usages  
    private Integer age;  
  
    public User(String name, Integer age) {  
        if (name == null || name.isBlank()) {  
            throw new IllegalArgumentException(String.format("잘못된 name(%s)이 들어왔습니다", name));  
        }  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() { return name; }  
  
    public Integer getAge() { return age; }  
  
}
```

# 하나씩 살펴봅시다!

JPA (Java Persistence API)

자바 진영의 ORM (Object-**Relational** Mapping)

관계형 DB의 테이블을 의미!

# 하나씩 살펴봅시다!

JPA (Java Persistence API)

자바 진영의 ORM (Object-**Relational** Mapping)

```
create table user
(
    id            bigint auto_increment,
    name          varchar(20) null,
    age           int         null,
    primary key (id)
);
```



# 하나씩 살펴봅시다!

JPA (Java Persistence API)

자바 진영의 ORM (Object-Relational Mapping)

둘을 짝짓는다!

# 하나씩 살펴봅시다!

JPA (Java Persistence API)

자바 진영의 ORM (Object-Relational Mapping)

```
public class User {  
  
    2 usages  
    private String name;  
    2 usages  
    private Integer age;  
  
    public User(String name, Integer age) {  
        if (name == null || name.isBlank()) {  
            throw new IllegalArgumentException(String.format("잘못된 name(%s)이 들어왔습니다", name));  
        }  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() { return name; }  
  
    public Integer getAge() { return age; }  
  
}
```

```
create table user  
(  
    id                bigint auto_increment,  
    name              varchar(20) null,  
    age               int         null,  
    primary key (id)  
);
```

# 하나씩 살펴봅시다!

JPA (Java Persistence API)

자바 진영의 ORM (Object-Relational Mapping)

```
public class User {  
  
    2 usages  
    private String name;  
    2 usages  
    private Integer age;  
  
    public User(String name, Integer age) {  
        if (name == null || name.isBlank()) {  
            throw new IllegalArgumentException(String.format("잘못된 name(%s)이 들어왔습니다", name));  
        }  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() { return name; }  
  
    public Integer getAge() { return age; }  
  
}
```

```
create table user  
(  
    id                bigint auto_increment,  
    name              varchar(20) null,  
    age               int         null,  
    primary key (id)  
);
```

# 총 정리 - JPA란?!

객체와 관계형 DB의 테이블을 짝지어  
데이터를 영구적으로 저장할 수 있도록 정해진 Java 진영의 규칙

# 총 정리 - JPA란?!

객체와 관계형 DB의 테이블을 짝지어  
데이터를 영구적으로 저장할 수 있도록 정해진 Java 진영의 **규칙**

**말로 되어 있는 규칙을 코드로 구현해야 한다!**



# JPA와 Hibernate

객체와 관계형 DB의 테이블을 짝지어  
데이터를 영구적으로 저장할 수 있도록 정해진 Java 진영의 규칙

↑ 구현 (implement)



구현체

# Hibernate는 내부적으로 JDBC를 사용한다!

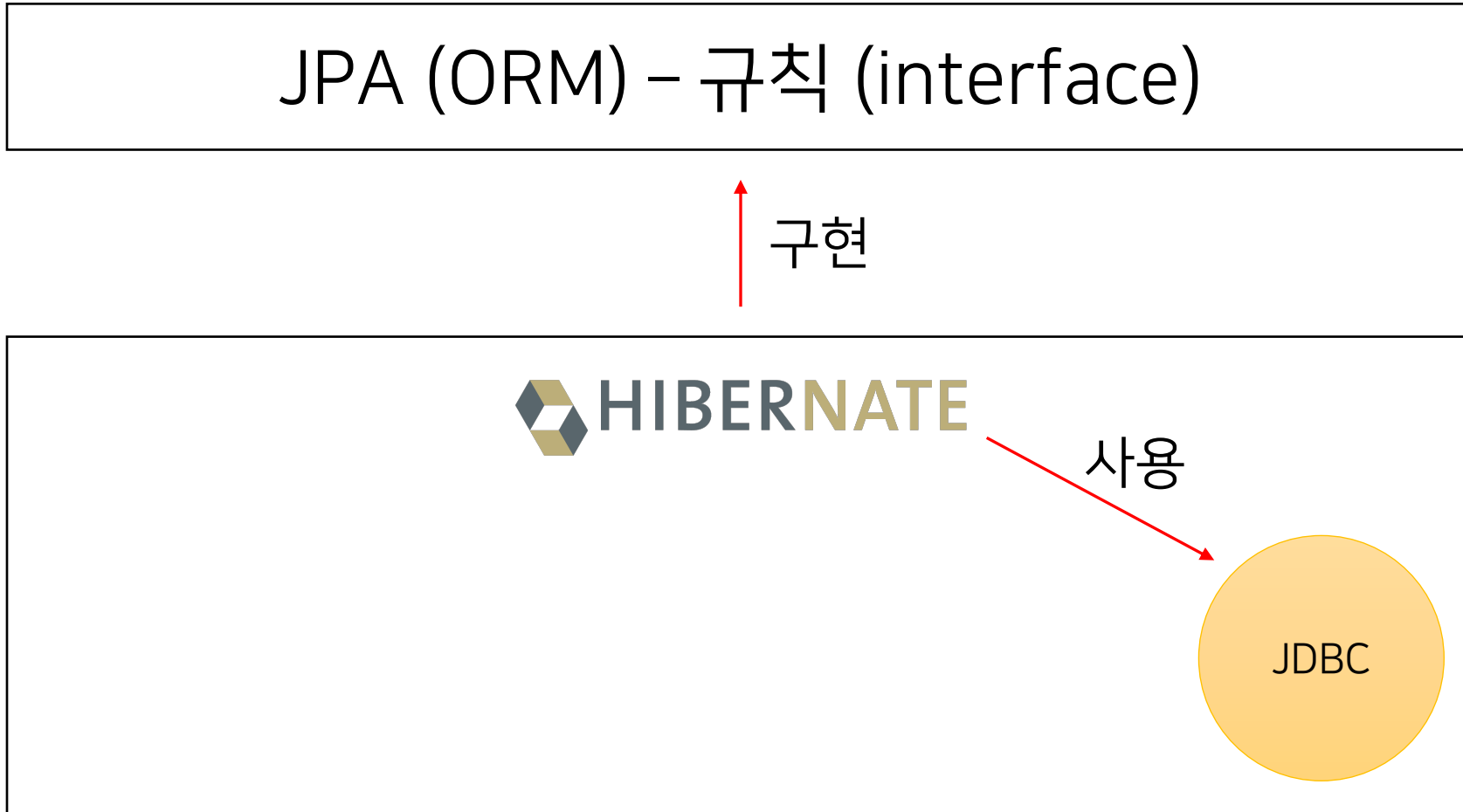
JPA (ORM) - 규칙 (interface)

↑  
구현

 HIBERNATE

사용

JDBC





**다음 시간에 JPA를 활용해 매핑을 해보자!**

# 24강. 유저 테이블에 대응되는 Entity Class 만들기

**Java객체와 MySQL Table을 매핑할 것이다!**

# 이전에 만들어두었던 User 객체를 활용하자!

```
public class User {  
  
    private String name;  
    private Integer age;  
  
    public User(String name, Integer age) {  
        if (name == null || name.isBlank()) {  
            throw new IllegalArgumentException(String.format("잘못된 name(%s)이 들어왔습니다", name));  
        }  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public Integer getAge() {  
        return age;  
    }  
}
```

# 이전에 만들어두었던 User 객체를 활용하자!

```
public class User {  
  
    private String name;  
    private Integer age;  
  
    public User(String name, Integer age) {  
        if (name == null || name.isBlank()) {  
            throw new IllegalArgumentException(String.format("잘못된 name(%s)이 들어왔습니다", name));  
        }  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public Integer getAge() {  
        return age;  
    }  
}
```

# 객체에는 name과 age가 있다!

```
public class User {  
  
    private String name;  
    private Integer age;  
}
```

# 우리가 만든 Table과 비교해보자!

```
public class User {  
    private String name;  
    private Integer age;
```

```
CREATE TABLE user  
(  
    id    bigint AUTO_INCREMENT,  
    name  varchar(20),  
    age   int,  
    PRIMARY KEY (`id`)  
);
```

**종습니다! 이제 시작해보겠습니다~!!**



# JPA 어노테이션

@Entity : 스프링이 User객체와 user 테이블을 같은 것으로 바라본다.

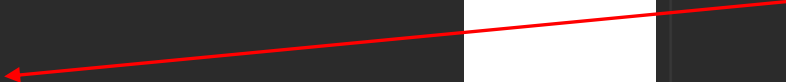
# Entity의 의미

저장되고, 관리되어야 하는 데이터

# 빠져 있는 id를 추가해주자!

```
public class User {  
    private String name;  
    private Integer age;
```

```
CREATE TABLE user  
(  
    id    bigint AUTO_INCREMENT,  
    name  varchar(20),  
    age   int,  
    PRIMARY KEY (`id`)  
);
```



# 빠져 있는 id를 추가해주자!

```
@Id  
@GeneratedValue(strategy = GenerationType.IDENTITY)  
private Long id = null;
```

# JPA 어노테이션

@Id : 이 필드를 primary key로 간주한다.

@GeneratedValue : primary key는 자동 생성되는 값이다.

# GeneratedValue

```
@Id  
@GeneratedValue(strategy = GenerationType.IDENTITY)  
private Long id = null;
```

DB의 종류마다 자동 생성 전략이 다르다!

우리는 MySQL의 auto\_increment를 사용했고,  
이는 IDENTITY 전략과 매칭된다.

# JPA를 사용하기 위해서는 기본 생성자가 꼭 필요하다!

```
protected User() {  
}
```

**id가 아닌 기본 column을 매핑하는 어노테이션**



# JPA 어노테이션

@Column : 객체의 필드와 Table의 필드를 매핑한다!

# User 객체의 name에 적용해보자

```
@Column(nullable = false, length = 20, name = "name")  
private String name;
```

# @Column

null이 들어갈 수 있는지 여부, 길이 제한,  
DB에서의 column 이름 등등...

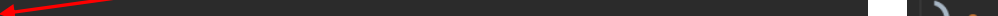
# @Column

사실 Column은 생략 할 수도 있다!

# 우리가 만든 Table과 비교해보자!

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private Long id = null;

@Column(nullable = false, length = 25)
private String name;
private Integer age;
```



```
CREATE TABLE user
(
    id    bigint AUTO_INCREMENT,
    name  varchar(20),
    age   int,
    PRIMARY KEY (`id`)
);
```

**이제 객체와 테이블의 매핑은 끝이 났다!!**

**단, JPA를 사용하니 추가적인 설정을 해주어야 한다.**

# application.yml

```
spring:
  jpa:
    hibernate:
      ddl-auto: none
    properties:
      hibernate:
        format_sql: true
        show_sql: true
        dialect: org.hibernate.dialect.MySQL8Dialect
```

스프링이 시작할 때 DB에 있는 테이블을 어떻게 처리할지



# **spring.jpa.hibernate.ddl-auto**

create : 기존 테이블이 있다면 삭제 후 다시 생성

create-drop : 스프링이 종료될 때 테이블을 모두 제거

update : 객체와 테이블이 다른 부분만 변경

validate : 객체와 테이블이 동일한지 확인

none : 별다른 조치를 하지 않는다.

# application.yml

```
spring:
  jpa:
    hibernate:
      ddl-auto: none
    properties:
      hibernate:
        format_sql: true
        show_sql: true
        dialect: org.hibernate.dialect.MySQL8Dialect
```

JPA를 사용해 DB에 SQL을 날릴 때 SQL을 보여줄 것인가

# application.yml

```
spring:
  jpa:
    hibernate:
      ddl-auto: none
    properties:
      hibernate:
        format_sql: true
        show_sql: true
        dialect: org.hibernate.dialect.MySQL8Dialect
```

SQL을 보여줄 때 예쁘게 포매팅 할 것인가

# application.yml

```
spring:
  jpa:
    hibernate:
      ddl-auto: none
    properties:
      hibernate:
        format_sql: true
        show_sql: true
        dialect: org.hibernate.dialect.MySQL8Dialect
```

dialect : 방언, 사투리

# application.yml

```
spring:
  jpa:
    hibernate:
      ddl-auto: none
    properties:
      hibernate:
        format_sql: true
        show_sql: true
        dialect: org.hibernate.dialect.MySQL8Dialect
```

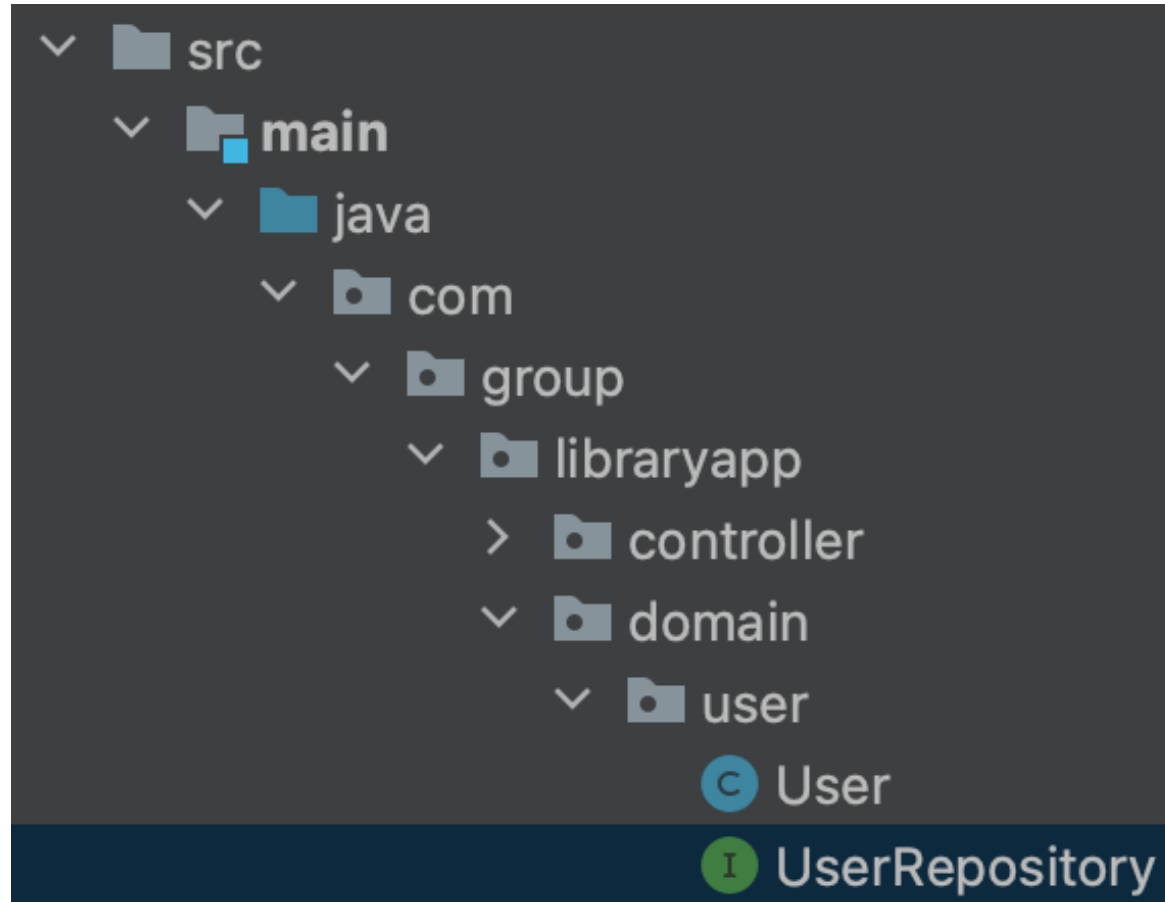
이 옵션으로 DB를 특정하면 조금씩 다른 SQL을 수정해준다.

# 25강. Spring Data JPA를 이용해 자동으로 쿼리 날리기

# 이번 시간에는..

SQL을 작성하지 않고,  
우리가 만들었던 유저 생성 / 조회 / 업데이트 기능을  
리팩토링 해볼 것이다.

# UserRepository 인터페이스를 User 옆에 만들어주자!





# 그 다음 JpaRepository를 상속 받아야 한다!

```
public interface UserRepository extends JpaRepository<User, Long> {  
  
}
```

**이제 JDBC를 활용하는 UserRepository 이름을 변경하자!**

# 유저 저장 기능

```
public void saveUser(UserCreateRequest request) {  
    userRepository.save(new User(request.getName(), request.getAge()));  
}
```

save 메소드에 객체를 넣어주면 INSERT SQL이 자동으로 날라간다!

# 유저 저장 기능

```
public void saveUser(UserCreateRequest request) {  
    userRepository.save(new User(request.getName(), request.getAge()));  
}
```

save 메소드에 객체를 넣어주면 **INSERT** SQL이 자동으로 날라간다!

# 유저 저장 기능

```
public void saveUser(UserCreateRequest request) {  
    userRepository.save(new User(request.getName(), request.getAge()));  
}
```

save되고 난 후의 User는 id가 들어 있다!

# 유저 조회 기능

```
public List<UserResponse> getUsers() {  
    return userRepository.findAll().stream() Stream<User>  
        .map(user -> new UserResponse(user.getId(), user.getName(), user.getAge()))  
        .collect(Collectors.toList());  
}
```

findAll을 사용하면 모든 데이터를 가져온다!  
**select \* from user;**

# 유저 조회 기능

```
public List<UserResponse> getUsers() {  
    return userRepository.findAll().stream() Stream<User>  
        .map(user -> new UserResponse(user.getId(), user.getName(), user.getAge()))  
        .collect(Collectors.toList());  
}
```

UserResponse에 생성자를 추가하면 코드가 깔끔해진다!

# 유저 조회 기능

```
public List<UserResponse> getUsers() {  
    return userRepository.findAll().stream() Stream<User>  
        .map(UserResponse::new) Stream<UserResponse>  
        .collect(Collectors.toList());  
}
```



# 유저 업데이트 기능

1. id를 이용해 User를 가져와 User가 있는지 없는지 확인하고
2. User가 있다면 update 쿼리를 날려 데이터를 수정한다.

# 유저 업데이트 기능

```
public void updateUser(UserUpdateRequest request) {  
    User user = userRepository.findById(request.getId())  
        .orElseThrow(IllegalArgumentException::new);  
  
    user.updateName(request.getName());  
    userRepository.save(user);  
}
```

# 유저 업데이트 기능

```
public void updateUser(UserUpdateRequest request) {  
    User user = userRepository.findById(request.getId())  
        .orElseThrow(InvalidArgumentException::new);  
  
    user.updateName(request.getName());  
    userRepository.save(user);  
}
```

findById를 사용하면 id를 기준으로  
1개의 데이터를 가져온다.

# 유저 업데이트 기능

```
public void updateUser(UserUpdateRequest request) {  
    User user = userRepository.findById(request.getId())  
        .orElseThrow(IllegalArgumentException::new);  
  
    user.updateName(request.getName());  
    userRepository.save(user);  
}
```

Optional의 orElseThrow를 사용해  
User가 없다면 예외를 던진다.

# 유저 업데이트 기능

```
public void updateUser(UserUpdateRequest request) {  
    User user = userRepository.findById(request.getId())  
        .orElseThrow(IllegalArgumentException::new);  
  
    user.updateName(request.getName());  
    userRepository.save(user);  
}
```

객체를 업데이트 해주고, save 메소드를 호출한다.  
그러면 자동으로 **UPDATE** SQL이 날라가게 된다.

# 지금까지 사용한 기능 정리

save : 주어지는 객체를 저장하거나 업데이트 시켜준다.

findAll : 주어지는 객체가 매핑된 테이블의 모든 데이터를 가져온다.

findById : id를 기준으로 특정한 1개의 데이터를 가져온다.

# 어떻게 SQL을 작성하지 않아도 동작하지?! JPA인가?!

반은 맞고, 반은 틀렸다!!

**어떻게 SQL을 작성하지 않아도 동작하지?! JPA인가?!**

Spring Data JPA



# Spring Data JPA

복잡한 JPA 코드를 스프링과 함께 쉽게 사용할 수 있도록  
도와주는 라이브러리

# Spring Data JPA

SimpleJpaRepository

# Spring Data JPA

Spring Data JPA

사용

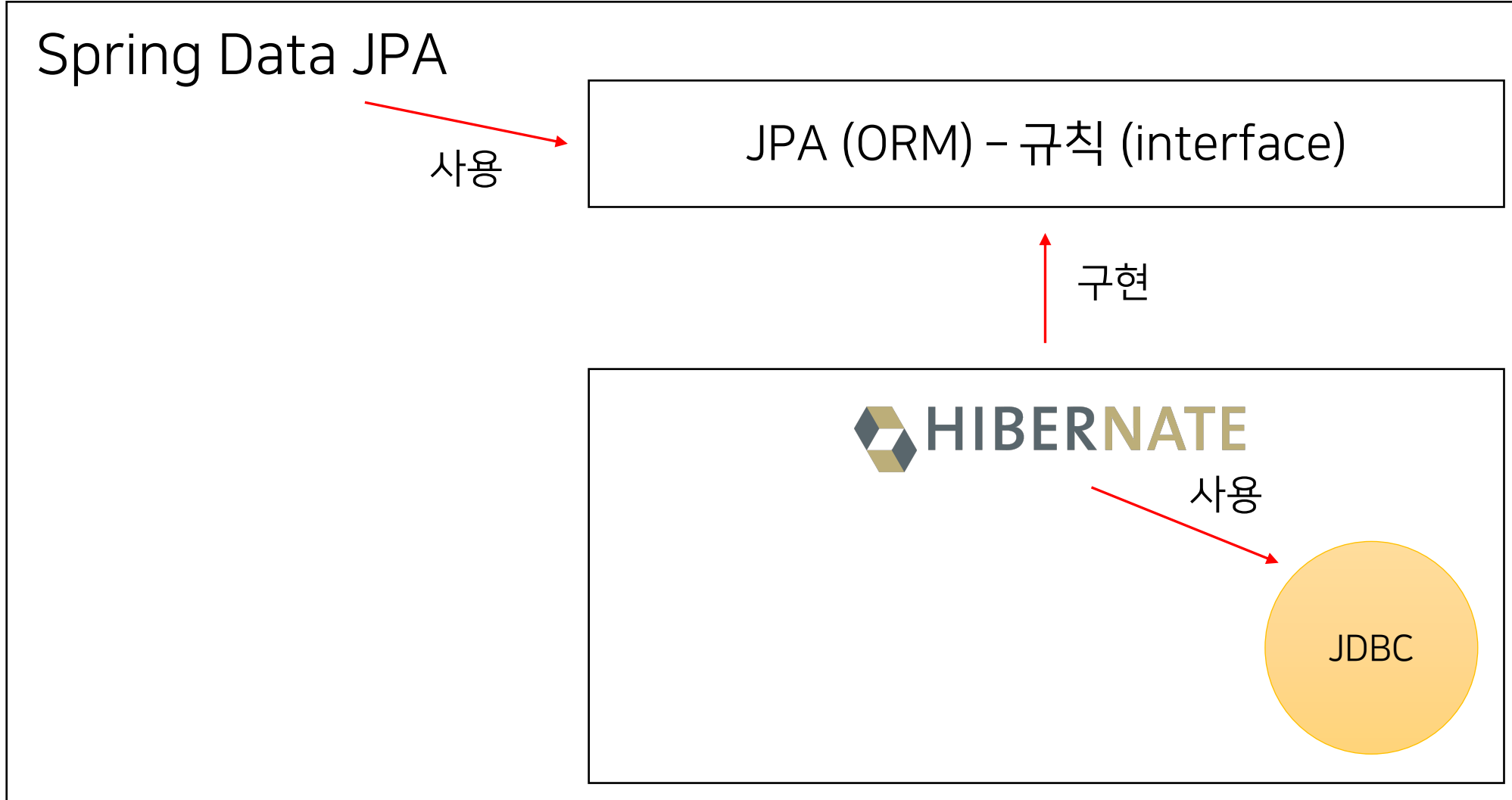
JPA (ORM) - 규칙 (interface)

구현

 HIBERNATE

사용

JDBC



Spring Data JPA
JPA (ORM)
Hibernate (JPA 구현체)
JDBC

# 26강. Spring Data JPA를 이용해 다양한 쿼리 작성하기

# 삭제 기능을 Spring Data JPA로 변경하자!

```
public void deleteUser(String name) {  
    if (userJdbcRepository.isUserNotExist(name)) {  
        throw new IllegalArgumentException();  
    }  
  
    userJdbcRepository.deleteUserByName(name);  
}
```

# 삭제 기능을 Spring Data JPA로 변경하자!

```
public void deleteUser(String name) {  
    if (userJdbcRepository.isUserNotExist(name)) {  
        throw new IllegalArgumentException();  
    }  
  
    userJdbcRepository.deleteUserByName(name);  
}
```

1) 이름을 기준으로 User가 있는지 확인하고

# 삭제 기능을 Spring Data JPA로 변경하자!

```
public void deleteUser(String name) {  
    if (userJdbcRepository.isUserNotExist(name)) {  
        throw new IllegalArgumentException();  
    }  
  
    userJdbcRepository.deleteUserByName(name);  
}
```

2) User가 있다면 DELETE 쿼리를 날린다.



# Spring Data JPA를 활용해 조회 쿼리를 어떻게 작성할까?

```
public interface UserRepository extends JpaRepository<User, Long> {  
  
    User findByName(String name);  
  
}
```

# Spring Data JPA를 활용해 조회 쿼리를 어떻게 작성할까?

```
public interface UserRepository extends JpaRepository<User, Long> {  
    User findByName(String name);  
}
```

반환 타입은 User이다. 유저가 없다면, null이 반환된다.

# Spring Data JPA를 활용해 조회 쿼리를 어떻게 작성할까?

```
public interface UserRepository extends JpaRepository<User, Long> {  
  
    User findByName(String name);  
  
}
```

함수 이름만 작성하면, 알아서 SQL이 조립된다!!

# Spring Data JPA를 활용해 조회 쿼리를 어떻게 작성할까?

```
public interface UserRepository extends JpaRepository<User, Long> {  
    User findByName(String name);  
}
```

find라고 작성하면, 1개의 데이터만 가져온다.

# Spring Data JPA를 활용해 조회 쿼리를 어떻게 작성할까?

```
public interface UserRepository extends JpaRepository<User, Long> {  
  
    User findByName(String name);  
  
}
```

By 뒤에 붙는 필드 이름으로  
SELECT 쿼리의 WHERE 문이 작성된다.

# Spring Data JPA를 활용해 조회 쿼리를 어떻게 작성할까?

```
public interface UserRepository extends JpaRepository<User, Long> {  
    User findByName(String name);  
}
```

SELECT \* FROM user WHERE name = ?;

# 삭제 기능 변경 완료!

```
public void deleteUser(String name) {  
    User user = userRepository.findByName(name);  
    if (user == null) {  
        throw new IllegalArgumentException();  
    }  
  
    userRepository.delete(user);  
}
```

주어지는 데이터를 DB에서 제거한다. (delete SQL)

**모든 기능을 바꾸었으니, 테스트 해보자!**



# 다양한 Spring Data JPA 쿼리

```
public interface UserRepository extends JpaRepository<User, Long> {  
  
    User findByName(String name);  
  
}
```

By 앞에 들어갈 수 있는 기능부터 살펴보자!

# By 앞에 들어갈 수 있는 구절 정리

find : 1건을 가져온다. 반환 타입은 객체가 될 수도 있고, Optional<타입>이 될 수도 있다.

findAll : 쿼리의 결과물이 N개인 경우 사용. List<타입> 반환.

exists : 쿼리 결과가 존재하는지 확인. 반환 타입은 boolean

count : SQL의 결과 개수를 센다. 반환 타입은 long이다.

# 다양한 Spring Data JPA 쿼리

```
public interface UserRepository extends JpaRepository<User, Long> {  
  
    User findByName(String name);  
  
}
```

이제 By 뒤에 들어갈 수 있는 기능들을 살펴보자!

각 구절은 And 나 Or로 조합할 수도 있다.

```
List<User> findAllByNameAndAge(String name, int age);
```

```
SELECT * FROM user WHERE name = ? AND age = ?;
```

# By 뒤에 들어갈 수 있는 구절 정리

GreaterThan : 초과

Between : 사이에

GreaterThanEqual : 이상

StartsWith : ~로 시작하는

LessThan : 미만

EndsWith : ~로 끝나는

LessThanEqual : 이하

# By 뒤에 들어갈 수 있는 구절 정리

```
List<User> findAllByAgeBetween(int startAge, int endAge);
```

```
SELECT * FROM user WHERE age BETWEEN ? AND ?;
```

**서비스 계층의 역할은 끝나지 않았다! - 트랜잭션**

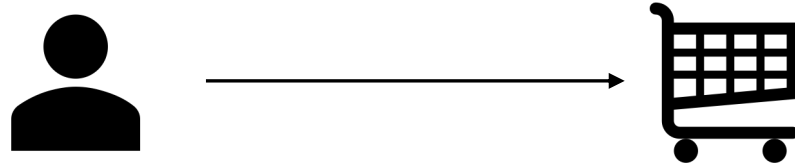
# 27강. 트랜잭션 이론편



# 트랜잭션이란?!

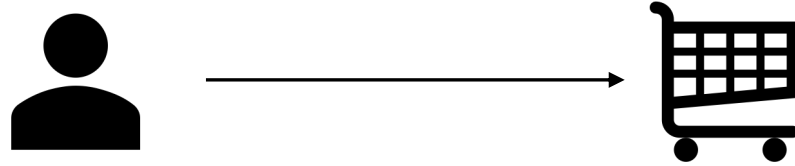
쪼갤 수 없는 업무의 최소 단위

# 쇼핑몰 사이트에서 물건을 주문하면..?!



- 1) 주문 기록을 저장하고
- 2) 포인트를 저장하고
- 3) 결제 기록을 저장해야 한다!

# 쇼핑몰 사이트에서 물건을 주문하면..?!




```
public class OrderService {  
  
    public void completePayment() {  
        orderRepository.save(new Order(..));  
        pointRepository.save(new Point(..));  
        billingHistoryRepository.save(new BillingHistory(..));  
    }  
  
}
```

# 자 그런데.. 만약 중간에서 예러가 난다면?!

```
public class OrderService {  
  
    public void completePayment() {  
        orderRepository.save(new Order(..));  
        pointRepository.save(new Point(..));  
        billingHistoryRepository.save(new BillingHistory(..));  
    }  
  
}
```


Error  
발생!



# 자 그런데.. 만약 중간에서 에러가 난다면?!

```
public class OrderService {  
  
    public void completePayment() {  
        orderRepository.save(new Order(..));  
        pointRepository.save(new Point(..));  
        billingHistoryRepository.save(new BillingHistory(..));  
    }  
  
}
```

Error  
발생!




주문 기록과 포인트는 있는데, 결제 기록이 없다!

# 자 그런데.. 만약 중간에서 에러가 난다면?!

```
public class OrderService {  
  
    public void completePayment() {  
        orderRepository.save(new Order(..));  
        pointRepository.save(new Point(..));  
        billingHistoryRepository.save(new BillingHistory(..));  
    }  
  
}
```

Error  
발생!



주문 기록은 있는데, 포인트와 결제 기록이 없다!

# 어떻게 이런 문제를 해결할 수 있을까?!

모든 SQL을 성공시키거나,  
하나라도 실패하면 모두 실패시키자!!

# 트랜잭션 시작하기

```
start transaction;
```



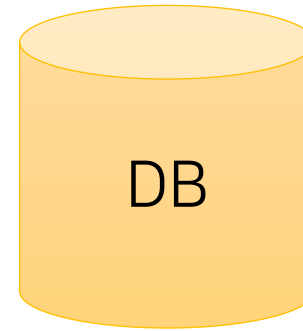
# 트랜잭션 정상 종료하기 (SQL 반영)

```
commit;
```

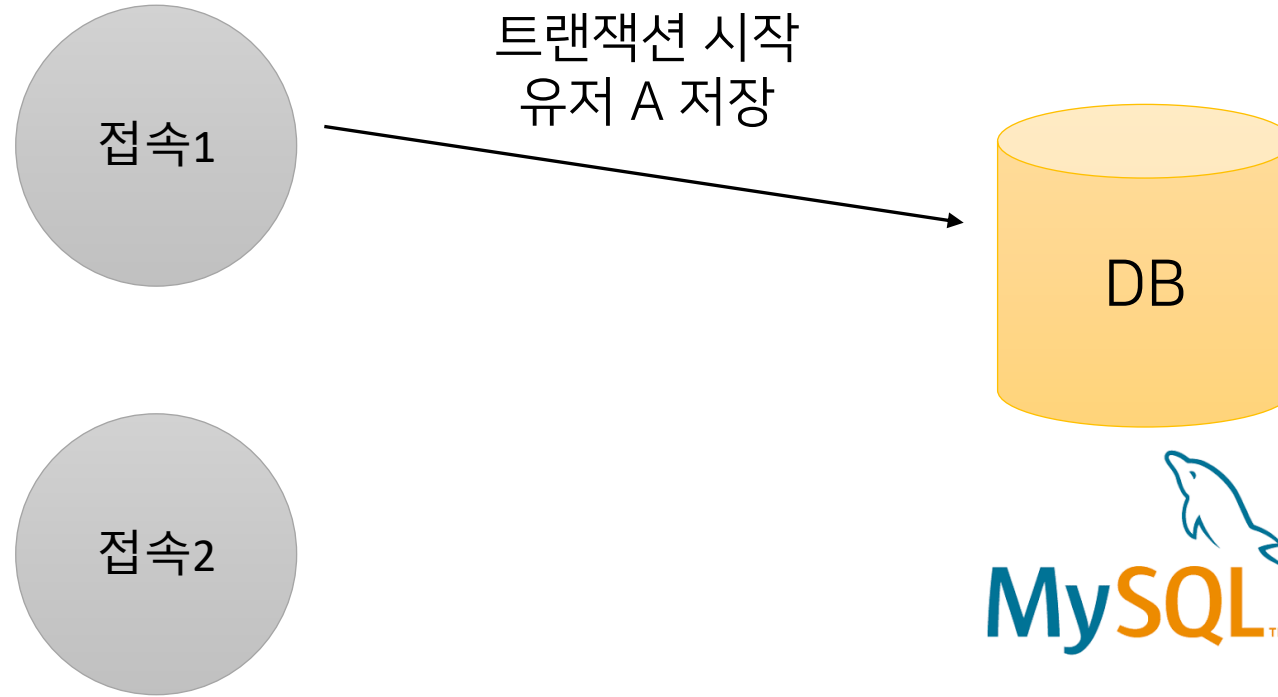
# 트랜잭션 실패 처리하기 (SQL 미반영)

```
rollback;
```

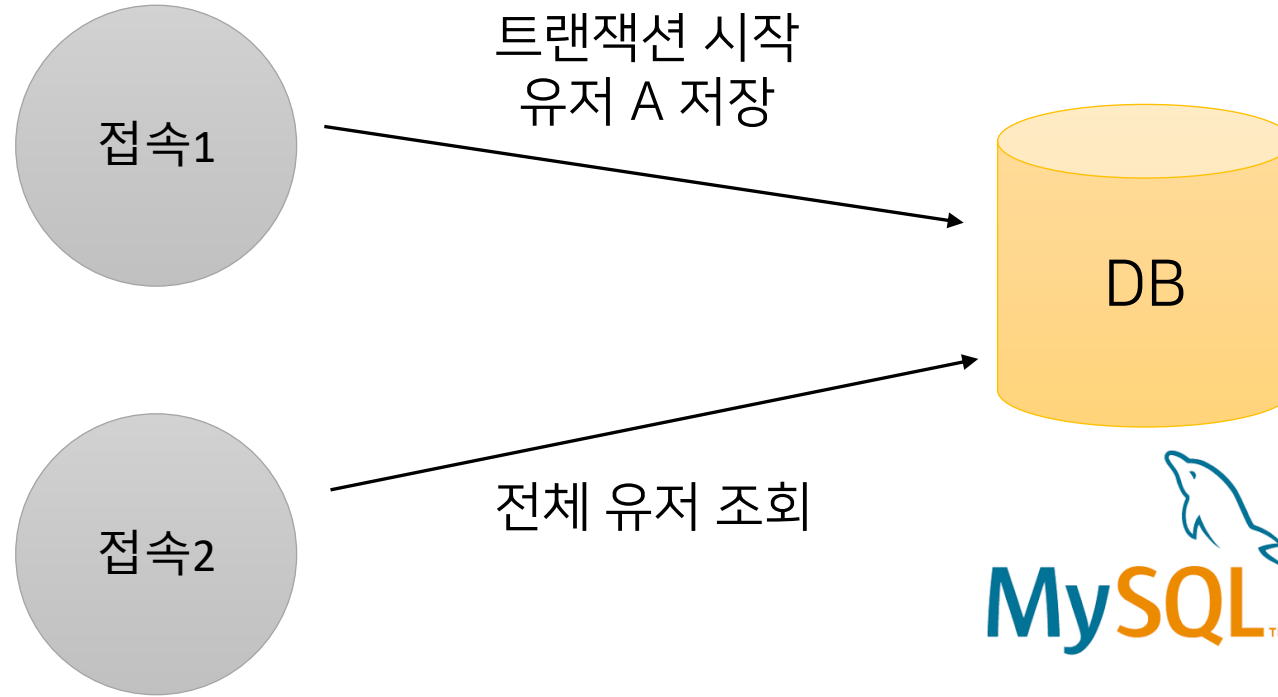
# 뭉어서 저장된다는 의미



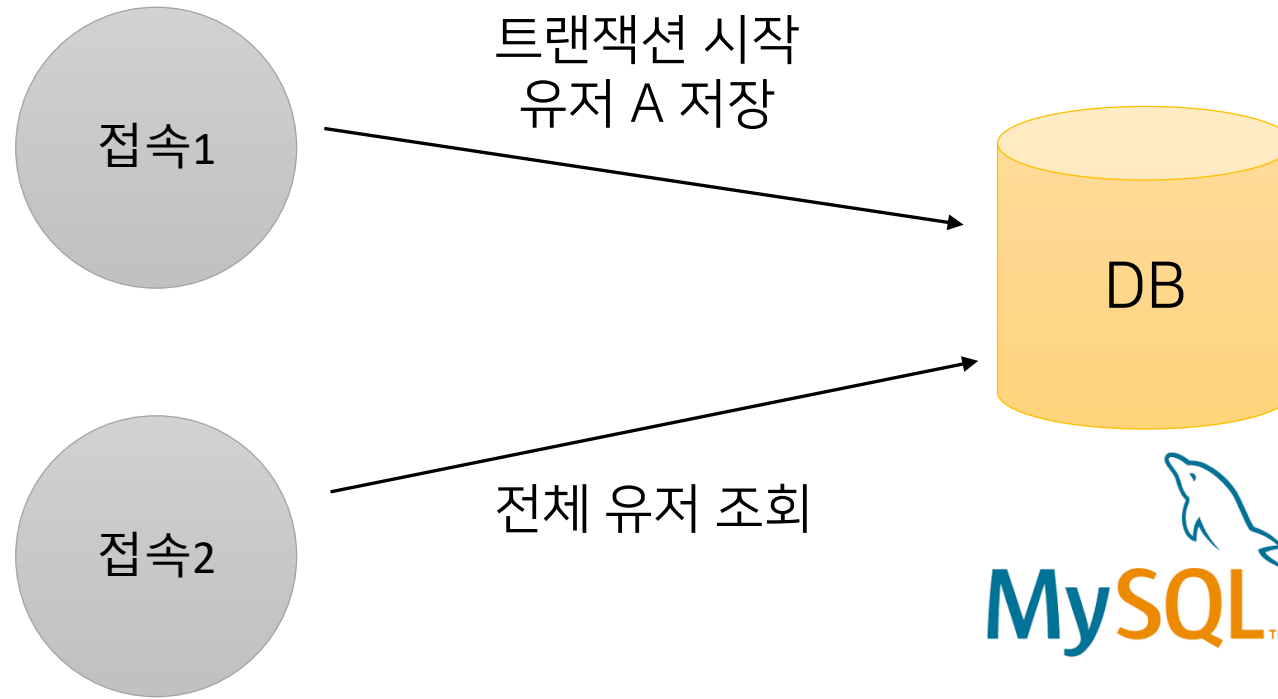
# 뭉어서 저장된다는 의미



# 묶어서 저장된다는 의미



# 뭉어서 저장된다는 의미



아직 트랜잭션 안의 SQL이 반영되지 않아,  
접속 2는 A 유저를 확인할 수 없다!!!

# 트랜잭션

쪼갤 수 없는 업무의 최소 단위  
= 모두다 성공시키거나 모두 다 실패시키거나

# 트랜잭션을 우리 코드에 어떻게 적용시킬 수 있을까?!

```
public class OrderService {  
  
    public void completePayment() {  
        orderRepository.save(new Order(..));  
        pointRepository.save(new Point(..));  
        billingHistoryRepository.save(new BillingHistory(..));  
    }  
  
}
```



# 28강. 트랜잭션 적용과 영속성 컨텍스트

# 우리가 원하는 것은...

1. 서비스 메소드가 시작할 때 트랜잭션이 시작되어
- 2-1. 서비스 메소드 로직이 모두 정상적으로 성공하면 commit되고
- 2-2. 서비스 메소드 로직 실행 도중 문제가 생기면 rollback 되는 것

**어떻게 트랜잭션을 적용할 수 있을까?!**

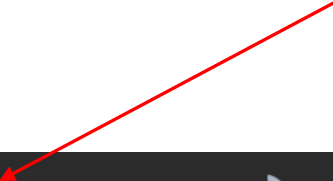
@Transactional

# 어노테이션으로 간단히 적용 가능하다.



```
@Transactional
public void saveUser(UserCreateRequest request) {
    userRepository.save(new User(request.getName(), request.getAge()));
}
```

# SELECT 쿼리만 사용한다면, readOnly 옵션을 쓸 수 있다!



```
@Transactional(readOnly = true)
public List<UserResponse> getUsers() {
    return userRepository.findAll().stream() Stream<User>
        .map(UserResponse::new) Stream<UserResponse>
        .collect(Collectors.toList());
}
```

# 주의 사항!

IOException과 같은  
Checked Exception은 롤백이 일어나지 않는다.

# 영속성 컨텍스트란?

테이블과 매핑된 Entity 객체를 관리/보관하는 역할

# 이것만 기억하면 됩니다!

스프링에서는 트랜잭션을 사용하면 영속성 컨텍스트가 생겨나고,  
트랜잭션이 종료되면 영속성 컨텍스트가 종료된다.



# 이것만 기억하면 됩니다!

영속성 컨텍스트 시작!



```
@Transactional
public void updateUser(UserUpdateRequest request) {
    User user = userRepository.findById(request.getId())
        .orElseThrow(IllegalArgumentException::new);

    user.updateName(request.getName());
    userRepository.save(user);
}
```

# 이것만 기억하면 됩니다!

```
@Transactional
public void updateUser(UserUpdateRequest request) {
    User user = userRepository.findById(request.getId())
        .orElseThrow(IllegalArgumentException::new);

    user.updateName(request.getName());
    userRepository.save(user);
}
```



영속성 컨텍스트 끝!

# 영속성 컨텍스트의 특수 능력 4가지

# [1] 변경 감지 (Dirty Check)

영속성 컨텍스트 안에서 불러와진 Entity는 명시적으로 save하지 않더라도, **변경을 감지해** 자동으로 저장된다.

# [1] 변경 감지 (Dirty Check)

```
@Transactional
public void updateUser(UserUpdateRequest request) {
    User user = userRepository.findById(request.getId())
        .orElseThrow(IllegalArgumentException::new);

    user.updateName(request.getName());
    userRepository.save(user);
}
```

# [1] 변경 감지 (Dirty Check)

```
@Transactional
public void updateUser(UserUpdateRequest request) {
    User user = userRepository.findById(request.getId())
        .orElseThrow(IllegalArgumentException::new);

    user.updateName(request.getName());
    userRepository.save(user);
}
```

# [1] 변경 감지 (Dirty Check)

```
@Transactional
public void updateUser(UserUpdateRequest request) {
    User user = userRepository.findById(request.getId())
        .orElseThrow(IllegalArgumentException::new);

    user.updateName(request.getName());
}
```

## [2] 쓰기 지연

DB의 INSERT / UPDATE / DELETE SQL을 바로 날리는 것이 아니라,  
트랜잭션이 commit될 때 모아서 한 번만 날린다.

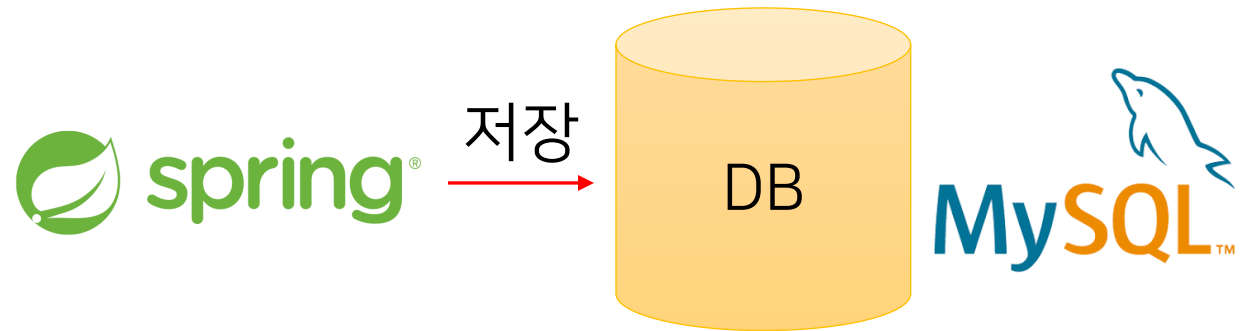


## [2] 쓰기 지연

```
@Transactional
public void saveUsers() {
    userRepository.save(new User(name: "A", age: 10));
    userRepository.save(new User(name: "B", age: 20));
    userRepository.save(new User(name: "C", age: 30));
}
```

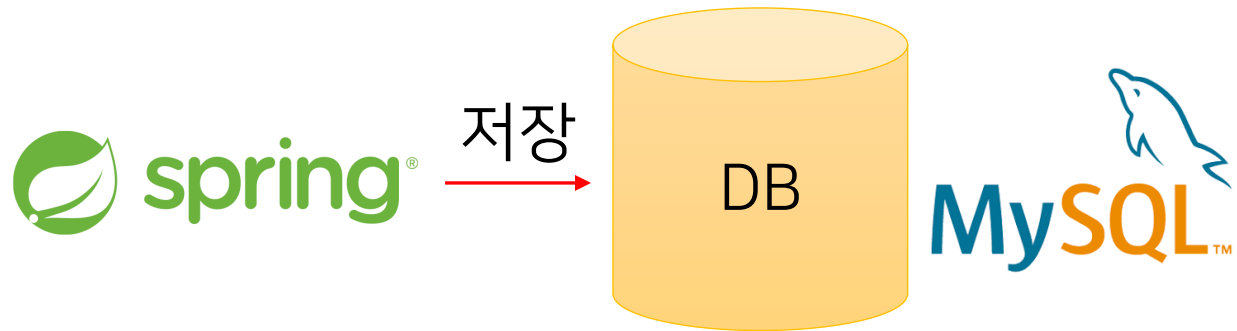
## [2] 쓰기 지연 - 쓰기 지연이 없다면?!

```
@Transactional
public void saveUsers() {
    userRepository.save(new User(name: "A", age: 10));
    userRepository.save(new User(name: "B", age: 20));
    userRepository.save(new User(name: "C", age: 30));
}
```



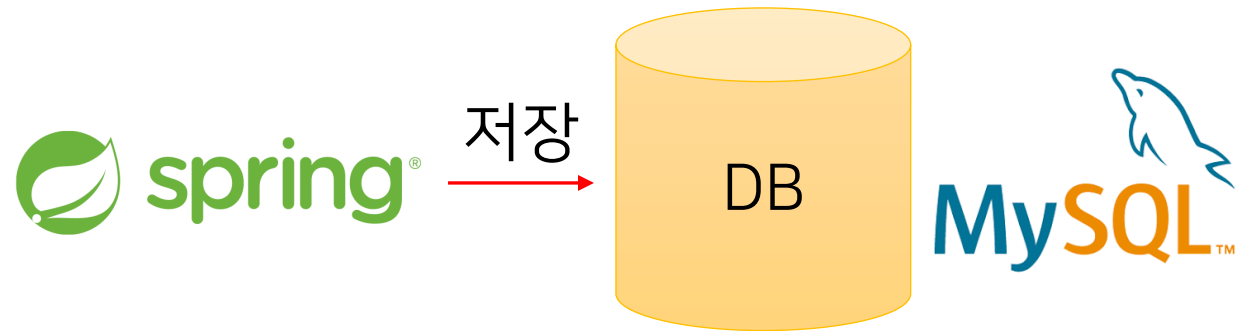
## [2] 쓰기 지연 - 쓰기 지연이 없다면?!

```
@Transactional
public void saveUsers() {
    userRepository.save(new User(name: "A", age: 10));
    userRepository.save(new User(name: "B", age: 20));
    userRepository.save(new User(name: "C", age: 30));
}
```



## [2] 쓰기 지연 - 쓰기 지연이 없다면?!

```
@Transactional
public void saveUsers() {
    userRepository.save(new User(name: "A", age: 10));
    userRepository.save(new User(name: "B", age: 20));
    userRepository.save(new User(name: "C", age: 30));
}
```



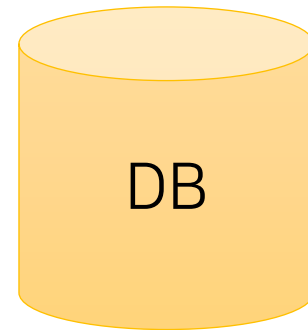
## [2] 쓰기 지연 - 쓰기 지연이 있다면?!

```
@Transactional
public void saveUsers() {
    userRepository.save(new User(name: "A", age: 10));
    userRepository.save(new User(name: "B", age: 20));
    userRepository.save(new User(name: "C", age: 30));
}
```



내가 기억했어!!

영속성 컨텍스트



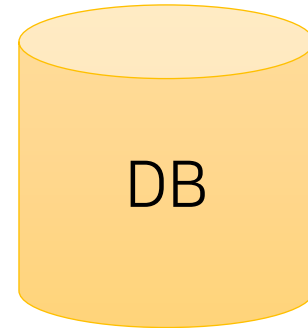
## [2] 쓰기 지연 - 쓰기 지연이 있다면?!

```
@Transactional
public void saveUsers() {
    userRepository.save(new User(name: "A", age: 10));
    userRepository.save(new User(name: "B", age: 20));
    userRepository.save(new User(name: "C", age: 30));
}
```



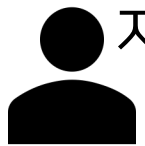
이것도 기억했어!!

영속성 컨텍스트



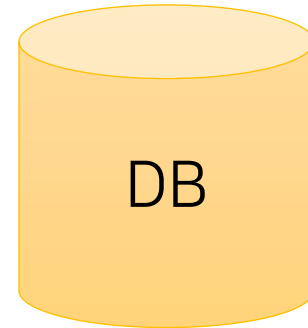
## [2] 쓰기 지연 - 쓰기 지연이 있다면?!

```
@Transactional
public void saveUsers() {
    userRepository.save(new User(name: "A", age: 10));
    userRepository.save(new User(name: "B", age: 20));
    userRepository.save(new User(name: "C", age: 30));
}
```



저것도 기억했어!!

영속성 컨텍스트



## [2] 쓰기 지연 - 쓰기 지연이 있다면?!

```
@Transactional
public void saveUsers() {
    userRepository.save(new User(name: "A", age: 10));
    userRepository.save(new User(name: "B", age: 20));
    userRepository.save(new User(name: "C", age: 30));
}
```





## [3] 1차 캐싱

ID를 기준으로 Entity를 기억한다!

## [3] 1차 캐싱

```
@Transactional(readOnly = true)
public void findUsers() {
    User user1 = userRepository.findById(1L).get();
    User user2 = userRepository.findById(1L).get();
    User user3 = userRepository.findById(1L).get();
}
```

## [3] 1차 캐싱

```
@Transactional(readOnly = true)
public void findUsers() {
    User user1 = userRepository.findById(1L).get();
    User user2 = userRepository.findById(1L).get();
    User user3 = userRepository.findById(1L).get();
}
```

## [3] 1차 캐싱

```
@Transactional(readOnly = true)
public void findUsers() {
    User user1 = userRepository.findById(1L).get();
    User user2 = userRepository.findById(1L).get();
    User user3 = userRepository.findById(1L).get();
}
```

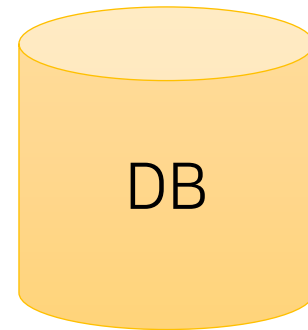


영속성 컨텍스트



spring®

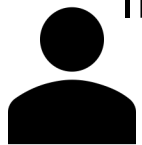
조회



## [3] 1차 캐싱

```
@Transactional(readOnly = true)
public void findUsers() {
    User user1 = userRepository.findById(1L).get();
    User user2 = userRepository.findById(1L).get();
    User user3 = userRepository.findById(1L).get();
}
```

User  
(ID=1)



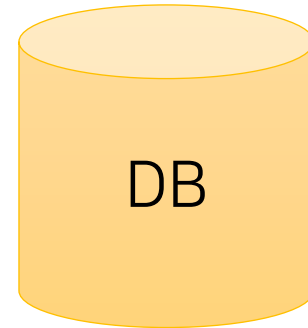
ID가 1인 유저 기억했어!!

영속성 컨텍스트



spring®

조회



MySQL™

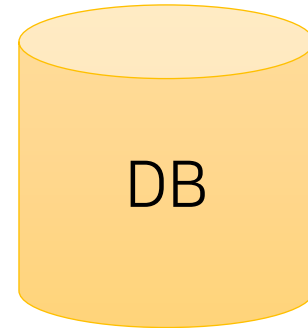
# [3] 1차 캐싱

```
@Transactional(readOnly = true)
public void findUsers() {
    User user1 = userRepository.findById(1L).get();
    User user2 = userRepository.findById(1L).get();
    User user3 = userRepository.findById(1L).get();
}
```

User  
(ID=1)



영속성 컨텍스트



# [3] 1차 캐싱

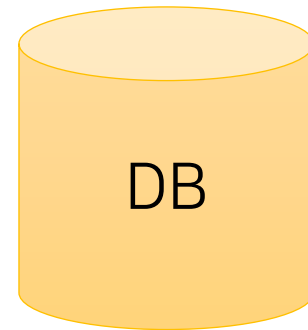
```
@Transactional(readOnly = true)
public void findUsers() {
    User user1 = userRepository.findById(1L).get();
    User user2 = userRepository.findById(1L).get();
    User user3 = userRepository.findById(1L).get();
}
```

User  
(ID=1)



ID가 1인 유저  
내가 알고 있어!

영속성 컨텍스트



# [3] 1차 캐싱

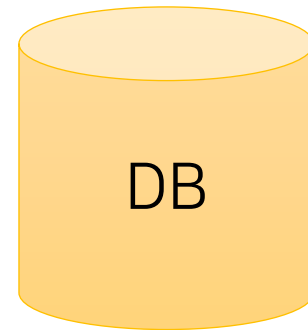
```
@Transactional(readOnly = true)
public void findUsers() {
    User user1 = userRepository.findById(1L).get();
    User user2 = userRepository.findById(1L).get();
    User user3 = userRepository.findById(1L).get();
}
```

User  
(ID=1)



ID가 1인 유저  
내가 알고 있어!

영속성 컨텍스트





## **[3] 1차 캐싱**

이렇게 캐싱된 객체는 완전히 동일하다!

**4번째 능력은 다음 Section에서!**

# 29강. Section 4 정리. 다음으로!

## Section 4. 생애 최초 JPA 사용하기

1. 문자열 SQL을 직접 사용하는 것의 한계를 이해하고,  
해결책인 JPA, Hibernate, Spring Data JPA가 무엇인지 이해한다.
2. Spring Data JPA를 이용해 데이터를 생성, 조회,  
수정, 삭제할 수 있다.

## Section 4. 생애 최초 JPA 사용하기

3. 트랜잭션이 왜 필요한지 이해하고,  
스프링에서 트랜잭션을 제어하는 방법을 익힌다.
4. 영속성 컨텍스트와 트랜잭션의 관계를 이해하고,  
영속성 컨텍스트의 특징을 알아본다.

**기본적인 준비물은 모두 끝이 났다!!!**

# 본격적으로 남은 요구사항을 구현해보자!

## 책

- 도서관에 책을 등록 및 삭제할 수 있다.
- 사용자가 책을 빌릴 수 있다.
  - 다른 사람이 그 책을 진작 빌렸다면 빌릴 수 없다.
- 사용자가 책을 반납할 수 있다.

**감사합니다**