섹션 4. 생애 최초 JPA 사용하기

이번 섹션의 목표

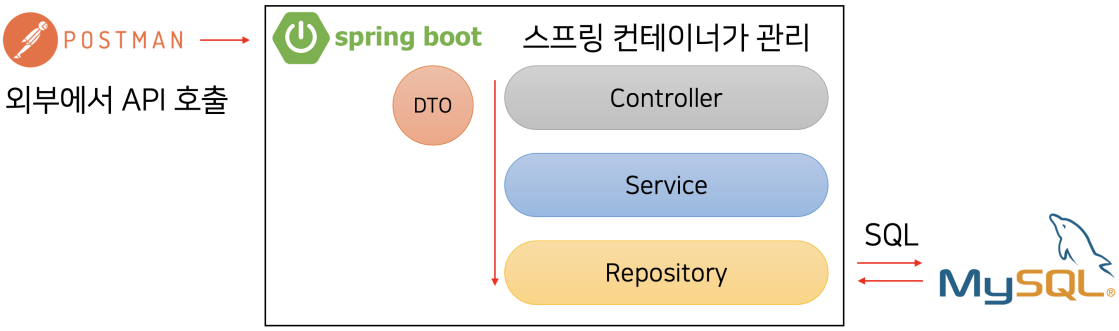
1. 문자열 SQL을 직접 사용하는 것의 한계를 이해하고, 해결책인 JPA, Hibernate, Spring Data JPA가 무엇인지 이해한다.
2. Spring Data JPA를 이용해 데이터를 생성, 조회, 수정, 삭제할 수 있다.
3. 트랜잭션이 왜 필요한지 이해하고, 스프링에서 트랜잭션을 제어하는 방법을 익힌다.
4. 영속성 컨텍스트와 트랜잭션의 관계를 이해하고, 영속성 컨텍스트의 특징을 알아본다.

23강. 문자열 SQL을 직접 사용하는 것이 너무 어렵다!!

강의 영상과 PPT를 함께 보시면 **더욱** 좋습니다 😊

이번 시간에는 우리가 작성했던 코드의 아쉬운 점을 살펴보고, 해결책을 정리해 보도록 하자! 우리가 지금까지 만든 방식을 살펴보면 다음과 같다.

지금까지 우리가 작성한 코드를 살펴보면...



여기서 우리는 DB에 접근하기 위해 SQL을 직접 작성했다. 하지만, 이런 방식으로 SQL을 직접 작성하는 것은 어딘가 아쉽다! 🥲 왜 SQL을 직접 작성하는 것이 아쉬울까?!

1. 문자열을 작성하기 때문에 실수할 수 있고, 실수를 인지하는 시점이 느리다.
    - 그나마 유료 개발 도구의 힘을 빌리면 SQL을 문자열로 작성할 때 오타가 나는 것을 IDE에서 확인할 수 있다. 하지만 사람마다 개발 환경이 다를 수도 있고, 실제 IDE에서 오타를 확인하지 못하는 경우에 실제 서버는 잘 실행되어 버린다. 이를 어려운 말로 '컴파일 타임' 때 발견하지 못하고 '런타임' 때 발견하는 오류라고 한다. 오류를 가진 채로 서버가 동작해서, 실제 기능 중에 말썽을 일으키는 것이다.
2. 특정 데이터베이스에 종속적이게 된다.
    - 사실 SQL도 데이터베이스의 종류마다 문법이 조금씩 다르다. MySQL, PostgreSQL, MSSQL 등등 모두 문법이 조금씩 다른데, 만약 수동으로 작성한 SQL이 1000개가 넘는데 데이터베이스를 바꿔야 한다면..?! 한 달 동안 집에 들어가지 못할 수도 있다!!!
3. 반복작업이 많아진다.
    - 테이블을 하나 만들 때마다 기본적인 INSERT / SELECT / UPDATE / DELETE 쿼리는 필요하다. 또한, SELECT 쿼리를 할 때마다 필드 하나하나를 매핑해주는 것은 무척 번거롭다.
4. 데이터베이스의 테이블과 객체는 패러다임이 다르다.
    - 대표적으로 연관관계와 상속이 있다. 예시로 연관관계를 Java 코드로 표현하면 다음과 같다.

```
// 교실 : 여러명의 학생을 가지고 있을 수 있다.
public class ClassRoom {
```

```
  private String name;
  private List<Student> students = new ArrayList<>();

  public ClassRoom(String name) {
    this.name = name;
  }

  public void addStudent(String name) {
    this.students.add(new Student(this, name));
  }

}

// 학생 : 어떤 교실에 소속된 학생인지 알 수 있다.
public class Student {

  private ClassRoom classRoom;

  private String name;

  public Student(ClassRoom classRoom, String name) {
    this.classRoom = classRoom;
    this.name = name;
  }

}

// main 함수
public static void main(String[] args) {
  ClassRoom classRoom = new ClassRoom("햇님반");
  classRoom.addStudent("A");
  classRoom.addStudent("B");
  classRoom.addStudent("C");
}
```

- ClassRoom도 Student를 가지고 있고, Student도 ClassRoom을 가지고 있다.
- 하지만 이를 Table로 표현하면 Student만 ClassRoom을 가리키게 된다.

교실 테이블	
id (PK)	이름 (varchar)
1	햇님반

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

그렇다, SQL을 직접 작성해 개발하게 되면 이런 어려움이 있었다. 그래서 사람들은 JPA라는 것을 만들게 되었다! JPA란, Java Persistence API의 약자로 자바 진영의

ORM(Object-Relational Mapping) 기술 표준을 의미한다. 우와~ 어려운 용어가 쏟아져 나왔다… 사실 우리가 모두 알고 있는 것이니 하나하나 살펴보자.

우선 Persistence, 한국어로 **영속성**이라는 개념부터 한 걸음씩 살펴보자. 우리가 Section 1에서 만들었던 서버를 기억해 보자. 가장 처음 만들었던 서버는 `List<User>`에 유저 정보를 저장했기 때문에 유저를 등록하더라도 서버가 종료되었다 켜지면 유저 정보가 모두 날아갔다! 그래서 우리는 데이터베이스를 사용해 유저 정보를 서버의 생사와 무관하게 저장하도록 했는데, 이것이 바로 **영속성**이다! 즉 데이터를 생성한 프로그램이 종료되더라도, 그 데이터는 영구적인 속성을 갖는 것이다.

다음 API라는 것은 우리가 만든 HTTP API에서도 쓰였지만, '정해진 규칙'을 의미한다.
그럼 여기까지 정리해 보면, JPA란 `데이터를 영구적으로 보관하기 위해 Java 진영에서 정해진 규칙`이다.

아직까지 아리송 하긴 하지만 좋다~ 이제 ORM (Object-Relational Mapping)이라는 의미를 살펴보자.

가장 먼저 Object라는 단어는 우리가 Java에서 사용하는 '객체'와 동일하다. 우리가 Step1에서 만들었던 User 클래스를 떠올려도 좋다.

```
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", na
me));
    }
    this.name = name;
    this.age = age;
  }

  public String getName() {
    return name;
  }

  public Integer getAge() {
    return age;
  }

}
```

다음으로 Relational이라는 의미는 관계형 데이터베이스의 '테이블'을 의미한다. 우리가 만든 user 테이블은 아래와 같다.

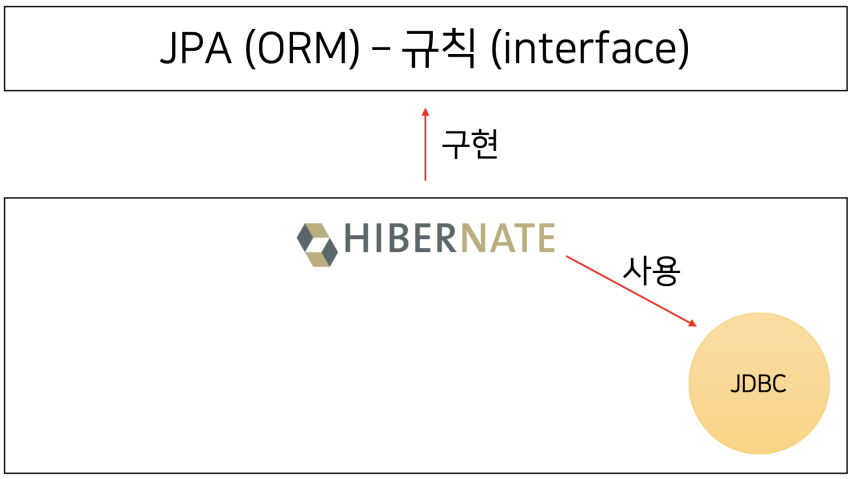
```
create table user(
  id bigint auto_increment,
  name varchar(25),
  age int,
  primary key (id)
)
```

마지막으로 Mapping이라는 의미는 말 그대로 둘을 짝지어 준다는 의미이다!! 우리가 코드로 작성한 User 객체를 MySQL에 있는 user 테이블과 짝지어준다고 생각하면 된다.

여기까지 정리하면 JPA란 다음과 같이 이해할 수 있다!

> 🔥 객체와 관계형 데이터베이스의 테이블을 짝지어
> 데이터를 영구적으로 저장할 수 있도록 정해진 Java 진영의 규칙

추가적으로 JPA는 API이기 때문에 말 그대로 '규칙'일뿐이고, 그 규칙을 실제 누군가는 코드로 작성해 주어야 한다. 마치 우리가 HTTP API를 만들 때 규칙을 말로 설명할 수 있지만, Controller를 이용해 그 규칙대로 동작할 수 있도록 실제 코딩을 한 것처럼 말이다. 이 JPA를 실제 코드로 작성한 가장 유명한 프레임워크가 바로 Hibernate이다. Hibernate은 내부적으로 JDBC를 사용하고 있다. 그림으로 나타내면 다음과 같다.



매우 좋다~! 👍 당연히 처음에는 100% 감이 오지 않을 수 있다. 다음 시간에는, 바로 한 번 우리가 만들었던 User라는 객체와 MySQL에 있는 user라는 테이블을 직접 매핑해보며 JPA를 알아보자!

24강. 유저 테이블에 대응되는 Entity Class 만들기

이번 시간에는 우리가 만들었던 유저 테이블에 매핑되는 객체를 직접 만들어보자. 이전에 우리가 개발해두었던 User 객체를 활용할 수 있다!

```
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", na
me));
    }
    this.name = name;
    this.age = age;
  }

  public String getName() {
    return name;
  }

  public Integer getAge() {
    return age;
  }

}
```

우선, 현재 User 객체와 DB에 존재하는 비교해 살펴보면,

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

`User` 객체에는 name, age 필드가 있고, `user` 테이블에는 id, name, age 필드가 존재한다.

좋다~!! 이제 시작해 보자.

가장 먼저 해주어야 할 것은, `User` 객체에 `@Entity`라는 어노테이션을 붙여주는 것이다. Entity는 '저장되고, 관리되어야 하는 데이터'를 의미한다.

어노테이션은 마법 같은 일을 해준다고 했다. `@Entity`를 붙이게 되면, 스프링이 이를 인식하여 서버가 동작할 때, `User` 객체와 `user` 테이블을 같은 것으로 간주할 것이다.

다음으로 해주어야 할 일은, `user` 테이블에만 존재하는 id를 `User` 객체에 추가해 주는 일이다. 이 id는 테이블에서 primary key이기 때문에 특별한 어노테이션을 붙여주어야 한다.

```
@Entity
public class User {

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

  // 생략...

}
```

- `@Id` : 이 필드를 primary key로 간주한다.
- `@GeneratedValue` : primary key는 DB에서 자동 생성해 주기 때문에 이 어노테이션을 붙여주어야 한다. DB의 종류마다 자동 생성 전략이 다른데, MySQL의 경우 `auto_increment`를 사용해 주고 있고, 이 전략은 `IDENTITY` 전략과 매칭된다.

매우 좋다~!! 👍 그런데 아직 User 클래스 아래 빨간 줄이 나오고 있다. 그 이유는, JPA에 의해 테이블과 매핑된 객체는 파라미터를 가지지 않은 기본 생성자가 꼭 필요하기 때문이다. 현재는 `User(String name, Integer age)`라는 파라미터를 2개 가진 생성자만 있기 때문에 에러가 나고 있다.

기본 생성자도 추가해 주자! protected도 해도 괜찮다!

```
@Entity
public class User {

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

  private String name;
```

```
  private Integer age;

  protected User() { }

  // 생략...
}
```

다음으로 기본적인 Column에 대해서도 어노테이션이 존재한다. 바로 `@Column` 어노테이션이다!

예를 들어 `private String name` 필드에는 다음과 같이 어노테이션을 붙일 수 있다.

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

`@Column` 어노테이션은 다양한 옵션이 있는데 주로 필드에 null이 들어갈 수 있는지의 여부나, 길이 제한, DB에서의 column 이름을 설정한다. 지금은 `User` 객체와 `user` 테이블의 필드 이름이 같지만, 다를 경우 `@Column` 어노테이션을 통해 설정해 주면 된다.

`@Column` 어노테이션이 존재하지 않더라도 JPA는 객체의 있는 필드가 Table에도 있을 거라 생각한다. 예를 들어, `private Integer age`라는 필드는 자동으로 `user` 테이블의 `age`와 매핑된다.

정말 좋다~!! 😊 모든 매핑이 끝이 났다! 이제 최초에 JPA를 적용할 때 설정해 주는 옵션을 추가해 주도록 하자. 이전에 데이터베이스 설정을 넣어두었던 `application.yml` 파일을 찾아 다음과 같이 입력해 주자.

```
spring:
  datasource:
    url: "jdbc:mysql://localhost/library"
    username: "root"
    password: ""
    driver-class-name: com.mysql.cj.jdbc.Driver
### 아래 부분이 추가되었다!!! ###
  jpa:
    hibernate:
      ddl-auto: none
    properties:
      hibernate:
        show_sql: true
        format_sql: true
        dialect: org.hibernate.dialect.MySQL8Dialect
```

하나씩 옵션의 의미를 살펴보면, 다음과 같다.

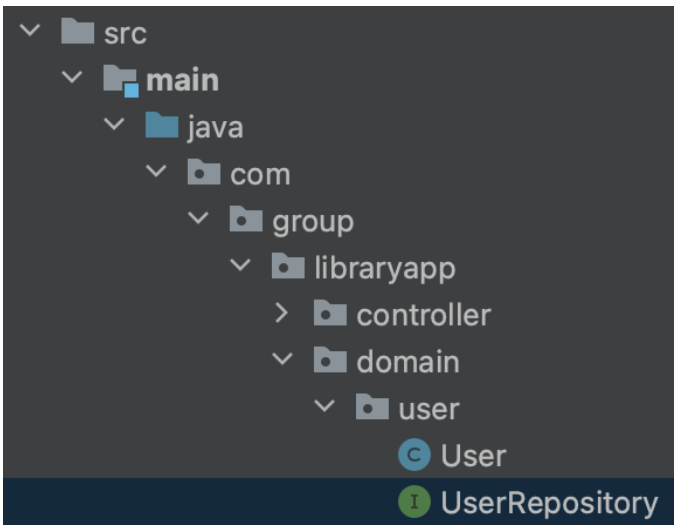
- spring.jpa.hibernate.ddl-auto
  - 스프링이 시작할 때 DB에 있는 테이블을 어떻게 처리할지에 대한 옵션이다.
    - create : 기존 테이블이 있다면 삭제 후 다시 생성한다.
    - create-drop : 스프링이 종료될 때 테이블을 삭제한다.
    - update : 객체와 테이블이 다른 부분만 변경한다.
    - validate : 객체와 테이블이 동일한지 확인한다.
    - none : 별다른 조치를 하지 않는다.
  - 현재 우리는 DB에 테이블이 잘 만들어져 있고, 미리 넣어둔 데이터도 있으므로 `none` 이라 설정해 주자!
- spring.jpa.properties.hibernate.show_sql
  - JPA를 사용해 DB에 SQL을 날릴 때 SQL을 보여줄지 결정한다.
  - 지금은 true로 설정해두자.
- spring.jpa.properties.hibernate.format_sql
  - JPA를 사용해 DB에 SQL을 날릴 때 SQL을 예쁘게 포맷팅할지 결정한다.
  - 마찬가지로 true로 설정해두도록 하자.
- spring.jpa.properties.hibernate.dialect
  - dialect란 한국어로 방언, 사투리라는 뜻이다.
  - 이 옵션을 통해 JPA가 알아서 Database끼리 다른 SQL을 조금씩 수정해 준다.
  - 우리는 MySQL 8버전을 사용하고 있으니 `org.hibernate.dialect.MySQL8Dialect`로 설정해 주면 된다.

이제 객체와 테이블 간의 매핑은 끝났다!!! 다음 시간에는 SQL을 직접 작성하지 않고 DB에 쿼리를 날려보자!

25강. Spring Data JPA를 이용해 자동으로 쿼리 날리기

이번 시간에는 SQL을 작성하지 않고 유저 테이블에 대해 쿼리를 날려, 우리가 만들었던 유저 생성 / 조회 / 업데이트 기능을 리팩토링할 것이다.

User 도메인 객체와 같은 위치에 `UserRepository`라는 인터페이스를 만들어 주자!




```
public interface UserRepository {

}
```

그런 다음, JpaRepository를 UserRepository가 상속받게 해주자. 이때, 우리가 만든 테이블의 매핑 객체인 `User` 와 유저 테이블의 id인 `Long` 타입을 각각 적어주어야 한다.

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

}
```

끝이다! 😊 SQL을 날릴 준비가 모두 끝났다! 이제 `UserService`에서 SQL을 이용해 직접 작성했던 이전 `UserRepository` 대신 새로운 `UserRepository`를 사용해 보자!

이전에 작성했던 `UserRepository`는 이름을 변경해 주자. `UserJdbcRepository`로 변경해두었다. 이름을 변경하지 않는다면, 현재 `UserRepository`가 2개이기 때문에 스프링이 시작될 때 오류가 날 것이다.

원래의 UserService 역시 `UserServiceV1`이라 이름을 변경하고, `UserServiceV2`를 새로 만들어 작업하자.

좋다! 👍 가장 먼저 `UserService`의 저장 기능부터 변경해 보자.

```
// JDBC 구현
public void saveUser(UserCreateRequest request) {
```

```
  userJdbcRepository.saveUser(request.getName(), request.getAge());
}
```

원래는 `UserRepository`에 이름과 나이를 보내고, `UserRepository`에 `INSERT` SQL을 작성해 유저 데이터를 저장해 주었다.

이제는 다음과 같이 적을 수 있다.

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

우리가 새로 만들었던 `UserRepository`에는 `save`라는 기능이 내장되어 있다. `User` 객체를 만들어 `save` 메소드를 호출하기만 하면 실제로 DB에 저장이 되는 것이다!! 정말 잘 되는지 한 번 확인해 보도록 하자. 콘솔에서 INSERT SQL이 잘 날라가는 것을 확인할 수 있다.

그리고, 이렇게 `save`를 하게 되면 저장된 `User` 객체가 반환되는데 이때 id를 사용할 수도 있다.

```
public void saveUser(UserCreateRequest request) {
  User user = userRepository.save(new User(request.getName(), request.getAge()));
  System.out.println(user.getId());
}
```

다음으로 조회 기능도 변경해 보자!

```
// JDBC 구현
public List<UserResponse> getUsers() {
  return userJdbcRepository.getUserResponses();
}
```

다음과 같이 변경할 수 있다

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

`findAll` 메소드를 사용하면 모든 유저 데이터를 조회하는 SQL이 날라가고, 그 결과 `List<User>`가 반환된다. 이 `List<User>`를 이제 `UserResponse`로 변경해 전달하는 것이다.

만약 `UserResponse`에서 `User`를 받는 생성자를 만든다면, 코드를 더욱 깔끔하게 변경할 수 있다.

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

다음으로는 업데이트 기능이다! 업데이트에서는 2번의 쿼리를 사용한다.

- id를 통해 User를 가져와 User가 있는지 없는지 확인하고
- User가 있다면 update 쿼리를 날려 데이터를 수정한다.

기존 코드는 다음과 같이 생겼다.

```
// JDBC 구현
public void updateUser(UserUpdateRequest request) {
  if (userJdbcRepository.isUserNotExist(request.getId())) {
    throw new IllegalArgumentException();
  }

  userJdbcRepository.updateUserName(request.getName(), request.getId());
}
```

이 코드를 변경하면 다음과 같다.

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

`findById`라는 기능을 사용해 id로 1개의 데이터를 가져올 수 있다. 이때 Java 라이브러리의 `Optional`이 반환되는데, `orElseThrow`를 사용하면 User가 비어있는 경우 에러를 던지게 된다.

그다음 반환된 `User` 객체의 이름을 업데이트해주고, 위에서 사용했던 `save` 기능을 호출하면 끝이다!!

이때 setter 대신 `updateName`이라는 명시적인 이름을 붙여준 이유는 아래 영상을 참고할 수 있다.

- https://www.youtube.com/watch?v=5P7OZceQ69Q

정말 좋다~ 😊 지금까지 우리가 사용했던 기능들을 정리해 보자!

- save : 주어지는 객체를 저장하거나 업데이트해준다.
- findAll : 주어지는 객체가 매핑된 테이블의 모든 데이터를 가져온다.
- findById : id를 기준으로 특정한 1개의 데이터를 가져온다.

자 그런데 한 가지 궁금한 게 생긴다. 어떻게 SQL을 작성하지 않아도 쿼리가 나갈 수 있을까?! 객체와 테이블을 자동으로 매핑해 준 JPA가 처리해 준 것일까?!

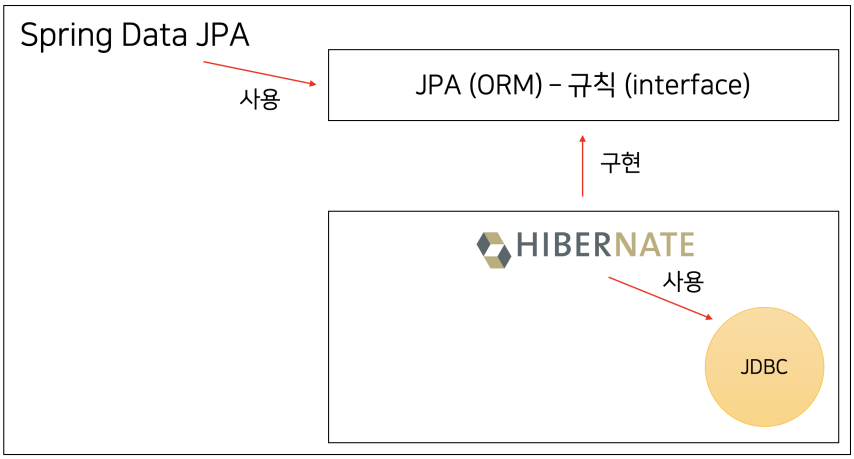
정답은, 비슷하지만 조금 다르다! JPA를 이용하는 `Spring Data JPA` 가 자동으로 처리해준 것이다. 23강에서 확인했던 JPA, Hibernate, JDBC 관계에 Spring Data JPA를 추가해 보면 다음과 같다.

우리가 사용했던 `save,` `findAll` 같은 메소드는 `SimpleJpaRepository`에서 찾아볼 수 있다. 스프링을 시작하면 여러가지 설정을 해준다고 했는데, 스프링은 `JpaRepository<Entity, ID>` 를 구현 받는 Repository에 대해 자동으로 `SimpleJpaRepository` 기능을 사용할 수 있게도 해준다.

`SimpleJpaRepository` 코드를 열어보면, 조금 복잡한 코드들을 확인할 수 있는데, 이게 바로 JPA 코드이다.

`Spring Data JPA`를 사용하는 덕분에 복잡한 JPA 코드를 직접 사용하는 게 아니라, 추상화된 기능으로써 사용할 수 있게 되는 것이다.

이를 그림으로 표현해 보면 다음과 같다.



매우 좋다~! 😊 이제 다음 시간에는 삭제 기능을 `Spring Data JPA` 를 활용해 변경해 보고, 다양한 조회 쿼리를 어떻게 작성할 수 있는지 알아보자!

26강. Spring Data JPA를 이용해 다양한 쿼리 작성하기

이번 시간에는 유저 삭제 기능을 구현해 보고, Spring Data JPA를 이용한 다양한 조회 쿼리 작성 방법을 알아보자.

현재 삭제 기능은 이렇게 구성되어 있다.

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

  userJdbcRepository.deleteUserByName(name);
}
```

업데이트 로직과 유사하게 이름을 이용하여 유저 존재 여부를 확인하고, 유저가 존재한다면, delete 쿼리를 날리게 된다.

자 우리는 여기서, '이름'을 기준으로 User 정보를 확인해야 한다! 이때 우리는 Spring Data JPA를 이용해 다양한 조회 쿼리를 작성할 수 있어야 한다.

UserRepository 인터페이스로 돌아가, 다음과 같은 메소드 시그니처를 작성해주자

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

  User findByName(String name);

}
```

- `User`
  - 이름을 기준으로 유저 데이터를 조회해 유저 객체를 반환한다.
  - 이때 유저 정보가 없다면, null이 반환된다.
- `findByName`
  - **함수 이름만 작성하면, 알아서 SQL이 조립된다.**
  - `find`라고 작성하게 되면, 1개의 데이터를 가져온다.
  - `By` 뒤에 붙는 필드 이름으로 SELECT 쿼리의 WHERE 문이 작성된다.
  - 예를 들어, `findByName`은 `select * from user where name = ?`이 된다.

매우 좋다~ 😊 삭제 기능을 마저 구현하고, Spring Data JPA를 사용하는 추가적인 규칙을 알아보자. 이제 이름을 기준으로 User 정보를 받아왔으니, `UserRepository`에 기본적으로 들어 있는 `delete` 메소드를 사용하면 된다.

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

  userRepository.delete(user);
}
```

이렇게, 생성 / 조회 / 업데이트 / 삭제 기능까지 모두 JDBC 대신 Spring Data JPA를 사용해 변경해 보았다.

UserController에서 UserServiceV2를 사용하게 변경하고 테스트해 보자. 이때 UserService 인터페이스를 만들어 다형성을 이용할 수도 있지만, 간단한 작업이니 객체 타입을 통째로 바꿔주자.

```
@RestController
public class UserController {

  // UserServiceV2를 사용하도록 변경
  private final UserServiceV2 userServiceV2;

  public UserController(UserServiceV2 userServiceV2) {
    this.userServiceV2 = userServiceV2;
  }

}
```

모든 기능이 잘 동작하는 것을 확인할 수 있다!! 👍

자 이제, Spring Data JPA의 추가적인 쿼리 작성법을 알아보자. 아까 언급했던 것처럼 By 앞과 뒤에 어떤 단어가 들어가는지에 따라 쿼리를 마음껏 만들어낼 수 있다.

By 앞에는 다음과 같은 구절이 들어갈 수 있다.

- find
  - 반환 타입은 객체가 될 수도 있고, `Optional<타입>`이 될 수도 있다.
- findAll
  - 쿼리의 결과물이 N개인 경우 사용한다. 반환 타입은 `List<타입>`이다.
- exists
  - 쿼리 결과가 존재하는지를 확인한다. 반환 타입은 `boolean`이다.
- count
  - SQL의 결과 개수를 센다. 반환 타입은 long이다.

By 뒤에는 필드 이름이 들어간다. 또한 이 필드들은 `And` 나 `Or`로 조합될 수 있다.

예를 들어, `findAllByNameAndAge`라 작성하게 되면, `select * from user name = ? and age = ?` 라는 쿼리가 나가게 되고, `findAllByNameOrAge`라 작성하게 되면, `select * from user name = ? or age = ?`라는 쿼리가 나가게 된다.

동등 조건 (`=`) 외에 다양한 조건을 활용할 수도 있다. 크다 작다를 사용할 수도 있고, 사이에 있는지 확인할 수도 있다. 또한 특정 문자열로 시작하는지 끝나는지 확인할 수도 있다.

- GreaterThan : 초과
- GreaterThanEqual : 이상
- LessThan : 미만
- LessThanEqual : 이하
- Between : 사이에
- StartsWith : ~로 시작하는
- EndsWith : ~로 끝나는

예를 들어 특정 나이 사이의 유저를 검색하고 싶다면, 다음과 같은 함수를 만들 수 있다.

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

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

}
```

절대 외울 필요 없다!! 그때그때 필요한 경우 찾아보다 보면, 자연스럽게 익숙해질 것이다~ 😊

자 이렇게! JPA와 Spring Data JPA를 활용하여 SQL을 직접 사용해야 하는 아쉬움을 해결했다.

하지만 아직 Service 계층의 역할은 끝나지 않았다! 서비스 계층의 중요한 역할이 하나 남아있는데, 바로 '트랜잭션' 관리이다. 다음 시간에는 트랜잭션이 무엇인지 그리고 왜 필요한지 알아보도록 하자.

27강. 트랜잭션 이론편

> 💡 강의 영상과 PPT를 함께 보시면 좋습니다 😊

이번 시간에는 트랜잭션이 무엇인지 그리고 왜 필요한지 살펴볼 예정이다.

트랜잭션을 구글에 검색하면, '쪼갤 수 없는 업무의 최소 단위'라는 설명이 나온다. 쪼갤 수 없다니 도대체 무슨 이야기일까?!

다음과 같은 경우 생각해 보자. 우리는 인터넷 쇼핑몰 사이트를 운영하고 있다. 우리의 인터넷 쇼핑몰에서, 물건을 주문하면, Service 계층에서는 다음과 같은 일을 처리해 주어야 한다.

결제가 완료되면, 1) 주문 기록을 쌓아주고 2) 포인트를 올려주고 3) 구매 기록을 저장해 준다. 간단히 코드로 느낌을 살려보면 다음과 같다.

```
public class OrderService {

  public void completePayment() {
    orderRepository.save(new Order(..));
```

```
    pointRepository.save(new Point(..));
    billingHistoryRepository.save(new BillingHistory(..));
  }

}
```

자 그런데, 문제가 발생한다. 주문 저장과 포인트 저장까지는 성공했는데, 구매 기록을 저장하는 과정에서 에러가 발생해 저장이 성공적으로 이루어지지 않은 것이다. 그럼 어떻게 될까?!!

`order` 테이블과 `point` 테이블에는 기록이 있는데, `billing_history` 테이블에는 기록이 없는 것이다!!! 도저히 말이 안 되는 것이다!

`point` 저장 과정에서 실패하더라도 마찬가지이다. `order` 테이블에만 기록이 있고 `point`와 `billing_history`에 기록이 없다면, 유저는 왜 포인트가 쌓이지 않았는지 문의를 접수할 것이다.

이런 문제를 해결하기 위해 '트랜잭션'이라는 개념이 등장한다. 여러 SQL을 사용해야 할 때 한 번에 성공시키거나, 하나라도 실패하면 모두 실패시키는 기능이다. 그래서 이것을 '쪼갤 수 없는 업무의 최소 단위'라고 표현하는 것이다.

아직까지 잘 와닿지 않을 수 있다. 직접 SQL을 이용해 트랜잭션을 직접 제어하며 이해해 보자. 2대의 컴퓨터가 DB에 접근해야 하므로, 2개의 console 창을 활용하거나 서로 다른 DB 접근법을 사용할 수 있다.

트랜잭션을 시작하려면 다음과 같은 명령어를 사용해야 한다.

```
start transaction;
```

트랜잭션이 시작되면, 트랜잭션을 성공적으로 종료하거나, 실패로 간주하기 전까지 SQL을 계속해서 날릴 수 있다.

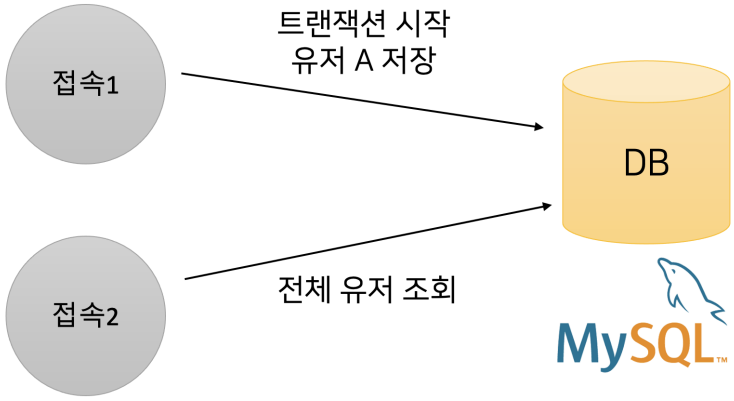
예를 들어, 2명의 유저를 한 트랜잭션으로 '묶어서' 저장해 보자.

```
start transaction;
insert into user (name, age) values ('A', 10);
insert into user (name, age) values ('B', 20);
commit;
```

트랜잭션이 시작된 후 사용된 SQL을 성공적으로 반영하려면 `commit`이라는 명령어를 사용해 주어야 한다.

자 그러면, 도대체 '묶어서' 저장한다는 것이 어떤 말일까?!

묶어서 저장된다는 의미



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

옵션을 주어 설정을 바꿀 수 있지만, 기본적으로 한 트랜잭션이 완료될 때까지는 그 트랜잭션 안에서 변경된 데이터가 보이지 않는다)

트랜잭션을 시작하고 사용한 SQL을 모두 취소하고 싶다면, `commit` 대신 `rollback`이라는 명령어를 사용하면 된다.

```
start transaction;
insert into user (name, age) values ('A', 10);
rollback; // rollback 했기 때문에, A 이름을 가진 유저가 user 테이블에 존재하지 않는다.
```

이처럼 트랜잭션을 통해 SQL을 묶어 관리하게 되면, 아까 같은 문제를 손쉽게 해결할 수 있다.

```
public class OrderService {

  public void completePayment() {
    orderRepository.save(new Order(..));
    pointRepository.save(new Point(..));
```

```
    billingHistoryRepository.save(new BillingHistory(..));
  }

}
```

`completePayment` 전체를 트랜잭션을 감싼 후, 3가지 데이터의 save가 정상적으로 수행되었다면 `commit`, 모종의 이유로 하나라도 실패했다면 `rollback` 을 하는 것이다.

그러면, `order` 데이터만 있고 `point` 와 `billing_history` 가 없는 일은 일어나지 않게 된다. 셋 모두 데이터가 존재하거나 셋 모두 데이터가 존재하지 않을 것이다.

매우 좋다~! 😊 이제 다음 시간에는 트랜잭션을 어떻게 적용할 수 있을지 알아보도록 하자!

28강. 트랜잭션 적용과 영속성 컨텍스트

이번 시간에는 트랜잭션을 `UserService`에 적용하고 JPA에 등장하는 영속성 컨텍스트라는 개념에 대해 알아보자.

지난 시간에 살펴보았던 것처럼, 우리가 원하는 것은

1. 서비스 메소드가 시작할 때 트랜잭션이 시작되어

2-1. 서비스 메소드 로직이 모두 정상적으로 성공하면 commit 되고

2-2. 서비스 메소드 로직 실행 도중 문제가 생기면 rollback 되는 것이다.

적용하는 방법은 매우 간단하다! 대상 메소드에 `@Transactional` 어노테이션을 붙여주면 된다. 끝이다!

주의할 점으로는 `org.springframework.transaction.annotation.Transactional` 을 붙여야 한다는 것이다. 다른 패키지의 `@Transactional`을 붙이면 정상 동작하지 않을 수 있다.

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

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

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

```
  userRepository.save(user);
}
```

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

  userRepository.delete(user);
}
```

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

데이터의 변경이 없고, 조회 기능만 있을 때는 `readOnly` 옵션을 줄 수 있다.

정말 끝이다! 😊 우리는 트랜잭션을 적용한 것이다~! 👍 정말 잘 적용했는지 간단하게 테스트를 해보자.

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

저장하는 로직에서 `IllegalArgumentException` 을 던지도록 해주었다. 트랜잭션이 잘 동작한다면, rollback이 일어나 우리가 원하는 유저 정보가 저장되지 않을 것이다. 서버를 재시작하고 웹 UI를 열어 간단히 확인해 보자!

`@Transactional` 어노테이션에 대해 한 가지 알아두어야 할 점은, Unchecked Exception에 대해서만 롤백이 일어난다는 점이다. IOException과 같은 Checked Exception에서는 롤백이 일어나지 않는다.

다음으로 영속성 컨텍스트에 대해 알아보자. 영속성 컨텍스트란, 테이블과 매핑된 Entity 객체를 관리/보관하는 역할을 수행한다. 음.. 어떤 느낌인지 감이 잘 오지 않는다!

지금은 이것만 기억하면 된다. 스프링에서는 트랜잭션을 사용하면 영속성 컨텍스트가 생겨나고, 트랜잭션이 종료되면 영속성 컨텍스트가 종료된다. 또한, 영속성 컨텍스트는 특별한 능력을 4가지 가지고 있다.

가장 먼저, `변경 감지 (Dirty Check)`라는 능력이다. 영속성 컨텍스트 안에서 불러와진 Entity는 명시적으로 `save`를 해주지 않더라도 알아서 변경을 감지하여 저장할 수 있게 해준다.

예를 들어, 현재의 업데이트 로직을 살펴보면, `User` 객체의 이름을 업데이트해준 후에, `save`를 직접 호출해 주고 있다. 하지만 이제 트랜잭션을 사용하고 있고, 트랜잭션이 시작되면 영속성 컨텍스트도 존재하기 때문에, 영속성 컨텍스트의 변경 감지 능력을 활용하면 `save`를 호출해 주지 않아도 괜찮다!

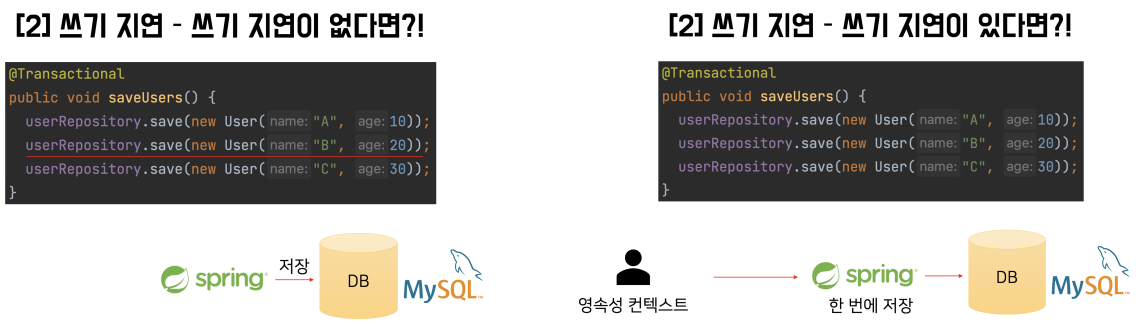
```
@Transactional
public void updateUser(UserUpdateRequest request) {
  User user = userRepository.findById(request.getId())
      .orElseThrow(IllegalArgumentException::new);

  user.updateName(request.getName());
  userRepository.save(user); // 생략 가능!!
}
```

다음으로는 `쓰기 지연` 능력이다. 이런 코드를 생각해보자!

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

이 코드는 마치 유저를 저장할 때마다 DB에 SQL을 날릴 것 같지만, 사실은 영속성 컨텍스트에 의해 트랜잭션이 commit 되는 시점에 SQL을 모아서 한 번만 날리게 된다. 이 기능을 '쓰기 지연'이라고 부른다. update나 delete 역시 마찬가지이다.



마지막으로는 `1차 캐싱` 능력이다. `1차 캐싱` 은 ID를 기준으로 Entity를 기억하는 기능이다.

```
@Transactional(readOnly = true)
public void saveUsers() {
  userRepository.findById(1L);
  userRepository.findById(1L);
  userRepository.findById(1L);
}
```

이 코드는 select 쿼리가 3번 날아갈 것 같지만, 영속성 컨텍스트에 의해 최초 1회만 select 쿼리가 나가게 된다. 나머지 2번은 영속성 컨텍스트가 보관하고 있는 데이터를 활용하게 된다.

이렇게 반환된 User는 완전히 동일하다.

```
System.out.println(user1 == user2); // true가 반환된다.
```

매우 좋다~! 👍👍 이렇게 이번 시간에는 트랜잭션을 `UserService`에 적용해 보고, 트랜잭션과 함께하는 영속성 컨텍스트의 특징에 대해서도 알아보았다. 다음 시간에는 이번 Section을 정리해 보도록 하자!

29강. Section 4 정리. 다음으로!

이번 Section을 거치며 우리는 문자열 SQL로 구성했던 우리의 데이터 접근 기술을 객체 지향 프로그래밍이 가능하도록 JPA를 활용해 완전히 변경했다!! 👍 또한, 이 과정에서 아래의 내용들을 익힐 수 있었다.

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

이제 기본적인 준비물은 모두 끝이 났다!! 이제 다음 시간에는 도서관리 애플리케이션의 남은 기능들과 함께 JPA의 연관관계에 대해 다루어 보도록 하자! 🔥 🏃