

3주차 - 데이터 베이스 연동

경희대학교 컴퓨터공학부 하계 리턴 백엔드(스프링부트) 스터디 3주차 - 트랙장
최현영

1. H2 및 Spring Data JPA 의존 라이브러리 추가

- H2 데이터베이스는 주로 개발용이나 소규모 프로젝트에서 사용되는 파일 기반의 경량 데이터베이스
 - 애플리케이션이 실행될 때, 해당 포트로 접속할 수 있기 때문에 종료된다면 데이터베이스도 같이 종료되며 기존에 저장된 데이터는 휘발된다.

```
dependencies {  
    ...  
    runtimeOnly 'com.h2database:h2'  
    implementation 'org.springframework.boot:spring-boot-starter-data-jpa'  
    ...  
}
```

설치한 H2 데이터베이스를 사용하기 위해서는 설정을 해야 한다. 다음과 같이 `application.yml` 파일을 수정하자.

[파일명: /returnStudy/src/main/resources/application.yml]

```
# DATABASE  
spring:  
  datasource:  
    driver-class-name: org.h2.Driver # 데이터베이스 접속시 사용하는 드라이버이다. Database를 H2로 사용하겠다는 것을 명시한다.  
    url: jdbc:h2:~/test # 데이터베이스 접속을 위한 경로이다.  
    username: sa # (사용자명은 기본 값인 sa로 설정한다.)  
    password: #로컬 개발 용도로만 사용하기 때문에 패스워드를 설정하지 않았다.  
  
  jpa:  
    hibernate:  
      ddl-auto: create  
    properties:  
      hibernate:  
        show_sql: true  
        format_sql: true  
  
  h2:  
    console:  
      enabled: true #H2 콘솔의 접속을 허용할지를 설정  
      path: /console #콘솔 접속을 위한 URL 경로이다.
```

2. ORM(Object Relational Mapping) ⇒ 객체 관계 맵핑을 의미

- a. 자바의 객체와 RDB의 테이블을 자동으로 1:1 매핑해준다.
- b. 자바의 클래스와 같은 객체지향 관점과 데이터베이스와 같은 데이터 중심 관점에서의 패러다임 불일치가 발생하는데 이를 해결한다.
 - i. 패러다임 불일치
 - 세분성 : 테이블 수와 엔티티 클래스의 수가 서로 다를 수 있다.
 - 상속성 : RDBMS에서는 상속이라는 개념이 없다
 - 식별성 : 데이터베이스는 기본키와 같은 식별자로 동일성을 정의하지만, 자바의 경우 두 객체의 값이 같을지라도 참조값이 다르다면 다르다고 판별할 수 있다.
 - 연관성 : 객체지향관점에서는 객체를 참조함으로써 연관성(의존성)을 나타내지만, RDBMS에서는 외래키만으로 연관성을 표현할 수 있다. 즉, 데이터베이스에서는 외래키만으로 양방향 연관관계를 표현할 수 있지만, 객체간에는 단방향만 가짐으로 불일치가 발생한다.
- c. ORM을 통해 쿼리문이 아닌 메서드로 데이터를 조작할 수 있다.
- d. ORM을 사용하면 내부에서 SQL 쿼리를 자동으로 생성해 주므로 직접 작성하지 않아도 된다. 즉, 자바만 알아도 데이터베이스에 질의할 수 있다.
- e. ORM을 이용하면 데이터베이스 종류에 상관 없이 일관된 코드를 유지할 수 있어서 프로그램을 유지·보수하기가 편리하다. 또한 내부에서 안전한 SQL 쿼리를 자동으로 생성해 주므로 개발자가 달라도 통일된 쿼리를 작성할 수 있고 오류 발생률도 줄일 수 있다.

3. JPA(Java Persistence API)

- a. JPA는 자바 진영에서 ORM(Object-Relational Mapping)의 기술 표준으로 사용하는 인터페이스의 모음
 - i. 즉, 자바진영에서의 ORM 표준 명세
- b. 실제로 동작하는 것이 아닌 어떻게 동작해야하는지 메커니즘을 정리한 명세서이다.
- c. 내부적으로 JDBC를 사용한다.
 - i. 기존에는 직접 SQL문을 작성하여 의존성이 높아 개발의 효율이 떨어짐
 - ii. JPA는 메서드를 통해 자동으로 SQL을 짜줌
- d. JPA의 대표적인 구현체인 하이버네이트를 주로 사용

4. Spring Data JPA

- a. Spring Data JPA는 하이버네이트를 조금 더 사용하기 쉽게 모듈화 해놓은 것
 - i. CRUD 처리에 필요한 인터페이스를 제공(메서드 제공)하며, 하이버네이트의 엔티티 매니저를 직접 다루지 않고 리파지토리를 정의해 사용함으로써 스프링이 알아서 적합한 쿼리를 대신 생성해줌
 - ii. 엔티티 매니저 : 엔티티를 관리하는 객체
- b. 우리는 Spring Data JPA를 통해 데이터베이스의 상태를 변화시켜 줄 것이다.

```
# DATABASE
spring:
  datasource:
    driver-class-name: org.h2.Driver
    url: jdbc:h2:~/test
    username: sa
    password:

  jpa:
    hibernate:
      ddl-auto: create
    properties:
      hibernate:
        show_sql: true
        format_sql: true

  h2:
    console:
      enabled: true
      path: /console
```

- spring.jpa.hibernate.ddl-auto : 엔티티를 기준으로 테이블을 생성하는 규칙을 정의한다.
 - none - 엔티티가 변경되더라도 데이터베이스를 변경하지 않는다.
 - update - 엔티티의 변경된 부분만 적용한다.

- validate - 변경사항이 있는지 검사만 한다.
- create - 스프링부트 서버가 시작될때 모두 drop하고 다시 생성한다.
- create-drop - create와 동일하다. 하지만 종료시에도 모두 drop 한다.
- spring.jpa.properties.show_sql : sql문을 출력
- spring.jpa.properties.format_sql : sql을 이쁘게 정렬

개발 환경에서는 보통 update 모드를 사용하고 운영환경에서는 none 또는 validate 모드를 사용한다.

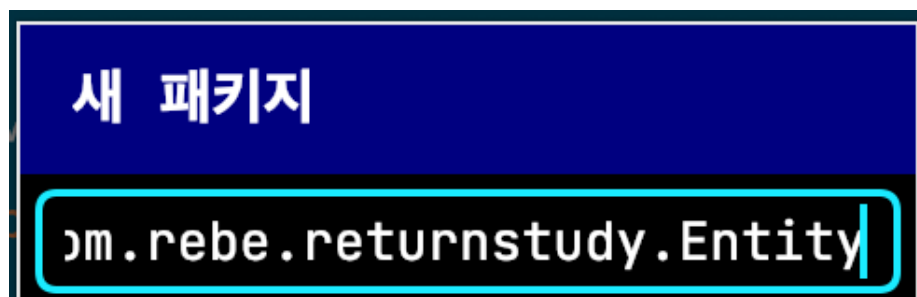
5. 롬복 익히기

- 롬복은 클래스를 생성할 때 반복적으로 사용하는 getter/setter, toString 같은 메서드를 어노테이션으로 대체하는 기능을 제공
- 롬복을 통해 코드를 짧게 구성할 수 있고 가독성을 높일 수 있다.
- 장점
 - 어노테이션 기반으로 코드를 자동으로 생성하여 생산성이 높아진다
 - 반복되는 코드를 생략할 수 있어 가독성이 높다
 - 간단하게 코드를 유추할 수 있어 유지보수에 용이하다.
- 어노테이션 종류
 - @Getter/ @Setter
 - 자동으로 각 필드의 getter/setter 메서드를 생성해줌
 - 생성자 자동 생성 어노테이션
 - @NoArgsConstructor : 매개변수가 없는 생성자를 자동으로 생성(기본 생성자와 같음)
 - @AllArgsConstructor : 모든 필드를 매개변수로 하는 생성자를 자동으로 생성
 - @RequiredArgsConstructor : 필드 중 final 접근 한정자(상수)로 설정된 필드를 갖는 생성자를 자동으로 생성함
 - @ToString

- Object 클래스의 메서드인 toString() 메서드를 자동으로 오버라이딩 해줌
- 해쉬값이 아닌, 각각의 필드값을 문자열로 반환해줌
- @EqualsAndHashCode
 - 객체의 동등성과 동일성을 비교하는 연산 메서드를 자동으로 생성
 - Object 클래스의 메서드인 hashCode와 equals 메서드를 자동으로 오버라이딩 해줌
 - 동등성 : 비교 대상이 되는 두 객체가 가진 내용물이 같음
 - 동일성 : 비교 대상이 되는 두 객체가 같은 객체
- @Data
 - 앞서 설명한 모든 어노테이션을 모두 포괄함
 - 이거 하나로 모든것이 해결됨
 - 권장하지 않음

5. 엔티티 설계

- JPA에서 엔티티는 데이터베이스의 테이블에 대응하는 클래스
- 데이터베이스에 쓰일 테이블과 칼럼을 정의함 ⇒ 어노테이션을 통해 연관관계를 정의할 수 있음[추후 Spring Data JPA와 연관관계 시간에 다룰 예정]
- 클래스 내의 필드는 테이블의 칼럼[애틀리뷰트]와 매핑됨



```
package com.rebe.returnstudy.Entity;

import jakarta.persistence.*;
import lombok.AllArgsConstructor;
```

```

import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Entity
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "member")
public class Member {
    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private Integer studentId;
    @Column(nullable = false)
    private String name;
    @Column(nullable = false)
    private String generation;
    @Column(nullable = false)
    private String club;
}

```

- application.yml에 정의한 ddl-auto의 값을 create로 설정해 주었기 때문에 엔티티를 자동으로 테이블로 변환하여 테이블을 생성함

6. 엔티티 관련 어노테이션

a. Entity

- 해당 클래스가 엔티티임을 명시함
- 해당 클래스와 테이블이 1:1 매칭되며, 인스턴스는 매핑되는 테이블에서 하나의 레코드를 의미함.

b. Table

- 클래스의 이름과 테이블의 이름을 다르게 지정해야할 경우 name이라는 옵션에 테이블명을 지정해 줄 수 있다.
- 없다면 클래스 이름이 그대로 테이블 이름으로 생성됨

c. Id

- 테이블의 기본값[PK] 역할로 사용됨 ⇒ 반드시 선언 되어야 함
- 모든 엔티티는 어떤 필드를 식별자[PK]로 할것인지 명시해 주어야함

d. GeneratedValue

- i. 해당 Id 필드를 어떤 방식으로 자동으로 생성할 것인지 결정할 때 사용
- ii. 직접 기본키를 할당해주고자 한다면 사용하지 않음

```
public enum GenerationType {  
  
    Indicates that the persistence provider must assign primary keys for the  
    entity using an underlying database table to ensure uniqueness.  
    TABLE,  
  
    Indicates that the persistence provider must assign primary keys for the  
    entity using a database sequence.  
    SEQUENCE,  
  
    Indicates that the persistence provider must assign primary keys for the  
    entity using a database identity column.  
    IDENTITY,  
  
    Indicates that the persistence provider must assign primary keys for the  
    entity by generating an RFC 4122 Universally Unique Identifier.  
    UUID,  
  
    Indicates that the persistence provider should pick an appropriate strategy  
    for the particular database. The AUTO generation strategy may expect a  
    database resource to exist, or it may attempt to create one. A vendor may  
    provide documentation on how to create such resources in the event that it  
    does not support schema generation or cannot create the schema resource at  
    the time of schema generation.  
    AUTO  
}
```

1. Auto

- a. 기본 설정 값
- b. 데이터베이스 시스템에 맞게 자동으로 생성함

2. Identity

- a. 기본값 생성을 전적으로 데이터베이스에 위임
- b. Auto_INCREMENT를 사용하여 기본값을 생성

e. Column

- i. 엔티티 클래스의 필드를 자동으로 테이블의 칼럼으로 매핑

```

public @interface Column {

    | (Optional) The name of the column. Defaults to the property or field name.
    |
    2개 구현
    String name() default "";

    | (Optional) Whether the column is a unique key. This is a shortcut for the
    | UniqueConstraint annotation at the table level and is useful for when the
    | unique key constraint corresponds to only a single column. This constraint
    | applies in addition to any constraint entailed by primary key mapping and to
    | constraints specified at the table level.
    |
    2개 구현
    boolean unique() default false;

    | (Optional) Whether the database column is nullable.
    |
    2개 구현
    boolean nullable() default true;

    | (Optional) Whether the column is included in SQL INSERT statements generated
    | by the persistence provider.
    |
    2개 구현
    boolean insertable() default true;

    | (Optional) Whether the column is included in SQL UPDATE statements generated
    | by the persistence provider.
    |
    2개 구현
    boolean updatable() default true;

    | (Optional) The SQL fragment that is used when generating the DDL for the
    | column.
    | Defaults to the generated SQL to create a column of the inferred type.
    |
    2개 구현
    String columnDefinition() default "";

    | (Optional) The name of the table that contains the column. If absent the
    | column is assumed to be in the primary table.
    |
    2개 구현
    String table() default "";

    | (Optional) The column length. (Applies only if a string-valued column is used.)
    |
    2개 구현
    int length() default 255;

```

1. name

- a. 데이터베이스의 칼럼명을 설정하는 속성
- b. 명시하지 않으면 필드명으로 생성됨

2. nullable

- a. 레코드를 생성할 때 컬럼 값이 null을 허용하는지 명시
- b. 기본값은 false이다.

3. length

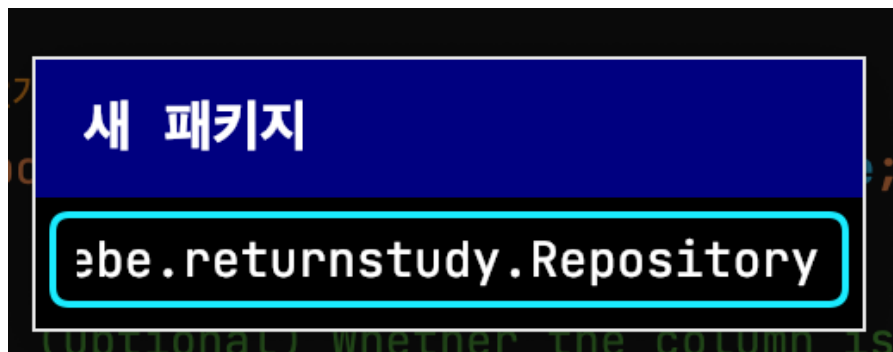
- a. 데이터의 최대 길이를 지정

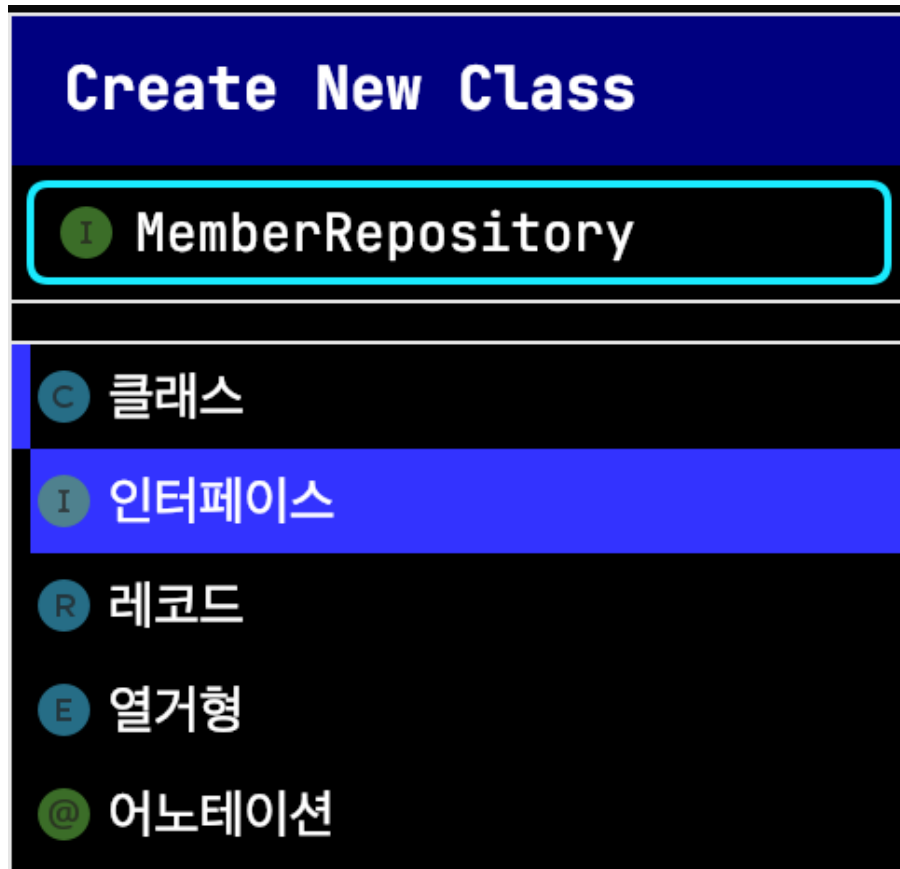
4. unique

- a. 해당 칼럼을 유니크로 설정[고유한 값, 중복 허용 x]

7. 리파지토리 인터페이스 생성

- a. Spring Data JPA는 **JpaRepository**를 기반으로 데이터베이스를 쉽게 사용할 수 있는 인터페이스를 제공
- b. 이를 상속하는 인터페이스를 이미 정의된 메서드를 쉽게 활용할 수 있다.
- c. 리파지토리 인터페이스는 리파지토리 패키지 내에서 생성된다.





```
package com.rebe.returnstudy.Repository;

import com.rebe.returnstudy.Entity.Member;
import org.springframework.data.jpa.repository.JpaRepository;

public interface MemberRepository extends JpaRepository<Member, Long> {
}
```

- JpaRepository를 상속받을 때는 대상 엔티티와 기본값의 타입을 지정해야한다.

```
public class Member {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```

앞서 엔티티 클래스 명을 Member로 하였고, 기본키 타입을 Long으로 선언하였기에 위와 같이 지정해 주었다.

```

public interface JpaRepository<T, ID> extends ListCrudRepository<T, ID>, ListPagingAndSortingRepository<T, ID>, QueryByExampleExecutor<T> {

    /**
     * Flushes all pending changes to the database.
     *
     * 1개 구현
     */
    void flush();

    /**
     * Saves an entity and flushes changes instantly.
     * 매개변수: entity - entity to be saved. Must not be null.
     * 반환: the saved entity
     *
     * 1개 구현
     */
    <S extends T> S saveAndFlush(S entity);

    /**
     * Saves all entities and flushes changes instantly.
     * 매개변수: entities - entities to be saved. Must not be null.
     * 반환: the saved entities
     * 시작 시간: 2.5
     *
     * 1개 구현
     */
    <S extends T> List<S> saveAllAndFlush(Iterable<S> entities);

    /**
     * Deletes the given entities in a batch which means it will create a single
     * query. This kind of operation leaves JPAs first level cache and the database
     * out of sync. Consider flushing the EntityManager before calling this method.
     * 지원 중단 Use deleteAllInBatch(Iterable) instead.
     * 매개변수: entities - entities to be deleted. Must not be null.
     *
     * @Deprecated
     */
    default void deleteInBatch(Iterable<T> entities) { deleteAllInBatch(entities); }

    /**
     * Deletes the given entities in a batch which means it will create a single
     * query. This kind of operation leaves JPAs first level cache and the database
     * out of sync. Consider flushing the EntityManager before calling this method.
     * 매개변수: entities - entities to be deleted. Must not be null.
     * 시작 시간: 2.5
     *
     * 3개 사용 위치 1개 구현
     */
    void deleteAllInBatch(Iterable<T> entities);
}

```

```

@Deprecated
T getOne(ID id);

Returns a reference to the entity with the given identifier. Depending on how
the JPA persistence provider is implemented this is very likely to always
return an instance and throw an jakarta.persistence.EntityNotFoundException on
first access. Some of them will reject invalid identifiers immediately.
지원 중단 use getReferenceById(ID) instead.
매개변수: id - must not be null.
반환: a reference to the entity with the given identifier.
시작 시간: 2.5
관련 주제: for details on when an exception is thrown.

1개 구현
@Deprecated
T getById(ID id);

Returns a reference to the entity with the given identifier. Depending on how
the JPA persistence provider is implemented this is very likely to always
return an instance and throw an jakarta.persistence.EntityNotFoundException on
first access. Some of them will reject invalid identifiers immediately.
매개변수: id - must not be null.
반환: a reference to the entity with the given identifier.
시작 시간: 2.7
관련 주제: for details on when an exception is thrown.

5개 사용 위치 1개 구현
T getReferenceById(ID id);

/*
 * (non-Javadoc)
 * @see org.springframework.data.repository.query.QueryByExampleExecutor#findAll(org.springframework.data.domain.Example)
 */
1개 구현
@Override
<S extends T> List<S> findAll(Example<S> example);

1개 구현
@Override
<S extends T> List<S> findAll(Example<S> example, Sort sort);

```

위와 같이 JpaRepository에는 많은 메서드가 정의되어 있다. 위의 메서드를 그대로 상속받았기 때문에 그대로 사용할 수 있다.

8. 영속성 컨텍스트 및 엔티티 매니저

a. 영속성 컨텍스트

- i. 엔티티 클래스와 테이블의 레코드의 패러다임 불일치를 해소하는 기능과 객체를 보관하는 기능을 수행
- ii. 엔티티 객체가 영속성 컨텍스트에 놓여지면, JPA는 엔티티 객체 내에 선언된 어노테이션을 바탕[매핑 정보]으로 데이터베이스에 반영 ⇒ 엔티티가 영속성 컨텍스트에 들어와 JPA가 관리하게 된다면 엔티티 객체는 **영속 객체**라고 불린다.
- iii. 영속 객체는 '영속'이라는 상태에 놓여진다.

1. 엔티티의 생명주기는 비영속, 영속, 준영속, 삭제로 구분되어 있는데 아래 기술 블로그를 참고 : <https://gmlwjd9405.github.io/2019/08/08/jpa-entity-lifecycle.html>

iv. 엔티티는 Repository 인터페이스의 메서드에 의해 영속성 컨텍스트에 놓여지게 되며, '영속 상태' 속에서 JPA가 데이터베이스와 매핑되도록 관리한다.

b. 엔티티 매니저

- i. 엔티티를 관리하는 객체
- ii. 데이터베이스에 접근해서 CRUD를 수행함
- iii. 우리가 방금 생성한 MemberRepository 인터페이스의 SUPER 클래스인 JpaRepository 내의 메서드를 실제로 구현한 구현체 클래스는 SimpleJpaRepository인데, 컴포넌트 시리즈인 @Repository 어노테이션과 데이터베이스를 CRUD 등의 작업을 하는 가장 최소 단위인 @Transactional 어노테이션이 선언되어 있으며, 내부 필드에 엔티티 매니저가 정의되어 있다.

```
@Repository
@Transactional(readOnly = true)
public class SimpleJpaRepository<T, ID> implements JpaRepositoryImplementation<T, ID> {

    4개 사용 위치
    private static final String ID_MUST_NOT_BE_NULL = "The given id must not be null";

    22개 사용 위치
    private final JpaEntityInformation<T, ?> entityInformation;
    33개 사용 위치
    private final EntityManager em;
    6개 사용 위치
    private final PersistenceProvider provider;

    18개 사용 위치
    private @Nullable CrudMethodMetadata metadata;
    9개 사용 위치
    private EscapeCharacter escapeCharacter = EscapeCharacter.DEFAULT;
```

8. 서비스 클래스 생성

- a. 서비스 레이어에서 데이터베이스의 접근을 수행함
 - i. 데이터베이스의 상태를 변경하는 것은 Repository 인터페이스가 수행
 - ii. 실제 비즈니스 로직[핵심기능]을 담당하는 서비스 클래스 생성

```

package com.rebe.returnstudy.Service;

import com.rebe.returnstudy.DTO.MemberDto;
import com.rebe.returnstudy.DTO.MemberResponseDto;
import com.rebe.returnstudy.Entity.Member;
import com.rebe.returnstudy.Repository.MemberRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.Optional;

@Service //서비스 레이어의 클래스에 사용하는 컴포넌트 시리즈 어노테이션
public class MemberService {

    private final MemberRepository memberRepository;

    @Autowired //의존성 주입
    public MemberService(MemberRepository memberRepository) {
        this.memberRepository = memberRepository;
    }

    //Create
    public MemberResponseDto saveMember(MemberDto memberDto) {
        Member member = new Member();
        member.setName(memberDto.getName());
        member.setStudentId(memberDto.getStudentId());
        member.setGeneration(memberDto.getGeneration());
        member.setClub(memberDto.getClub());

        memberRepository.save(member);
        //save 메서드를 통해 엔티티를 데이터베이스에 저장
        MemberResponseDto memberResponseDto = new MemberResponseDto();
        memberResponseDto.setId(member.getId());
        memberResponseDto.setName(member.getName());
        memberResponseDto.setStudentId(member.getStudentId());
        memberResponseDto.setClub(member.getClub());
        memberResponseDto.setGeneration(member.getGeneration());

        return memberResponseDto;
    }

    //Read
    public MemberResponseDto getMember(Long id) {
        Optional<Member> member = memberRepository.findById(id);
        //findById 메서드를 통해 pk값으로 해당 엔티티를 가져온다.
        if (member.isPresent()) {
            MemberResponseDto memberResponseDto = new MemberResponseDto();
            memberResponseDto.setId(member.get().getId());
            memberResponseDto.setStudentId(member.get().getStudentId());
            memberResponseDto.setName(member.get().getName());
            memberResponseDto.setGeneration(member.get().getGeneration());
            memberResponseDto.setClub(member.get().getClub());

            return memberResponseDto;
        }
    }
}

```

```

        } else {
            return null;
        }
    }

    //Update
    public MemberResponseDto updateMember(Long id, MemberDto memberDto) {
        Optional<Member> member = memberRepository.findById(id);
        if (member.isPresent()) {
            Member updateMember = member.get();
            updateMember.setStudentId(memberDto.getStudentId());
            updateMember.setClub(memberDto.getClub());
            updateMember.setGeneration(memberDto.getGeneration());
            updateMember.setName(memberDto.getName());
            //update라는 메서드는 존재하지 않으며, 우선, findById을 통해 해당 엔티티를 영속 객체로 둔다
            //단지 영속성 컨텍스트에 존재하는 영속 객체와 새로 갱신한 엔티티를 비교하여(dirty check)하여
            //변경을 감지하고 해당 필드[에트리뷰트]만 갱신하는 update 쿼리가 발생한다.
            memberRepository.save(updateMember);

            MemberResponseDto memberResponseDto = new MemberResponseDto();
            memberResponseDto.setId(updateMember.getId());
            memberResponseDto.setStudentId(updateMember.getStudentId());
            memberResponseDto.setName(updateMember.getName());
            memberResponseDto.setGeneration(updateMember.getGeneration());
            memberResponseDto.setClub(updateMember.getClub());

            return memberResponseDto;
        } else {
            return null;
        }
    }

    //Delete
    public boolean isDeleteMember(Long id){
        Optional<Member> member = memberRepository.findById(id);
        if (member.isPresent()) {
            memberRepository.delete(member.get());
            return true;
        } else {
            return false;
        }
    }
}

```

- 리파지토리 패키지[레이어]에 정의된 클래스를 의존성 주입하여 의존 메서드를 호출하도록 하였다.
- JpaRepository 인터페이스 내, 사전 정의된 메서드를 호출하여 데이터베이스의 상태를 변경하도록 하였다.

b. 요청 바디에 사용할 Dto 수정

```
package com.rebe.returnstudy.DTO;

@NoArgsConstructor
@AllArgsConstructor
@ToString
@Getter
@Setter
public class MemberDto {
    Integer studentId;
    String name;
    String generation;
    String club;
}
```

d. 응답 바디에 사용할 Dto 생성

```
package com.rebe.returnstudy.DTO;

import lombok.*;

@NoArgsConstructor
@AllArgsConstructor
@ToString
@Getter
@Setter
public class MemberResponseDto {
    private Long id;
    private Integer studentId;
    private String name;
    private String generation;
    private String club;
}
```

8. 컨트롤러 클래스 생성

- 비즈니스 로직과 클라이언트의 요청을 연결하는 컨트롤러 생성
- 컨트롤러는 클라이언트로 부터 요청을 받고, 해당 요청을 서비스 레이어에 구현된 서비스 클래스의 적절한 메서드를 호출하여 결과값을 단지 클라이언트에게 반환만 한다.

```
package com.rebe.returnstudy.Controller;

import com.rebe.returnstudy.DTO.MemberDto;
import com.rebe.returnstudy.DTO.MemberResponseDto;
```



```

import com.rebe.returnstudy.Service.MemberService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/return")
public class MemberController {

    private final MemberService memberService;

    @Autowired //핵심 로직을 처리하는 서비스 클래스를 의존성 주입한다.
    public MemberController(MemberService memberService){
        this.memberService = memberService;
    }

    @GetMapping(value = "/member/{id}")
    public ResponseEntity<MemberResponseDto> getStudentInfo(@PathVariable(value = "id") Long id ) {
        System.out.println("회원 번호 : " + id);
        MemberResponseDto memberResponseDto = memberService.getMember(id);

        if(memberResponseDto == null){
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).build();
        }
        else{
            return ResponseEntity.status(HttpStatus.OK).body(memberResponseDto);
        }
    }

    @PostMapping("/registration")
    public ResponseEntity<MemberResponseDto> registerMember(@RequestBody MemberDto memberDto){
        System.out.println(memberDto.toString());
        return ResponseEntity.status(HttpStatus.CREATED).body(memberService.saveMember(memberDto));
    }

    @PutMapping("/update/{id}")
    public ResponseEntity<MemberResponseDto> registerMember(@PathVariable(value = "id") Long id,
                                                            @RequestBody MemberDto memberDto){

        System.out.println(memberDto.toString());
        MemberResponseDto memberResponseDto = memberService.updateMember(id, memberDto);

        if(memberResponseDto == null){
            return ResponseEntity.status(HttpStatus.BAD_REQUEST).build();
        }
        else{
            return ResponseEntity.status(HttpStatus.ACCEPTED).body(memberResponseDto);
        }
    }

    @DeleteMapping("/withdraw/{id}")
    public ResponseEntity<Void> dropOutMember(@PathVariable(value = "id") Long id){
        if(memberService.isDeleteMember(id)){
            return ResponseEntity.ok().build();
        }
    }
}

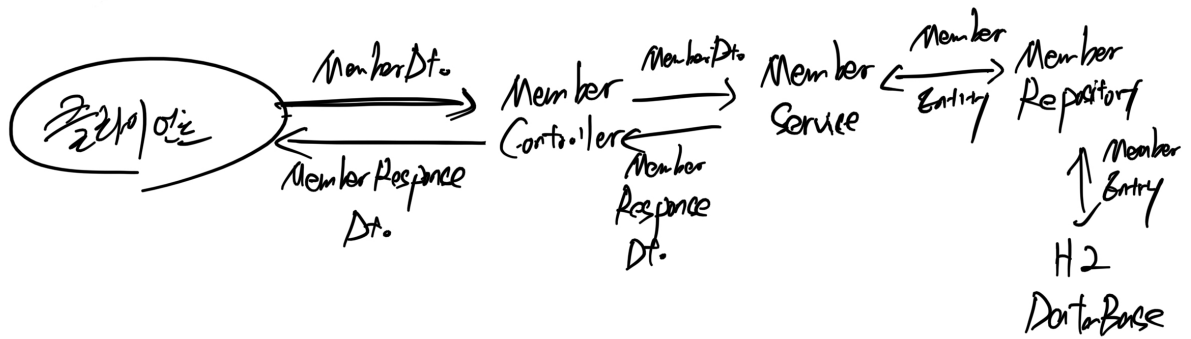
```

```

else{
    return ResponseEntity.badRequest().build();
}
}
}

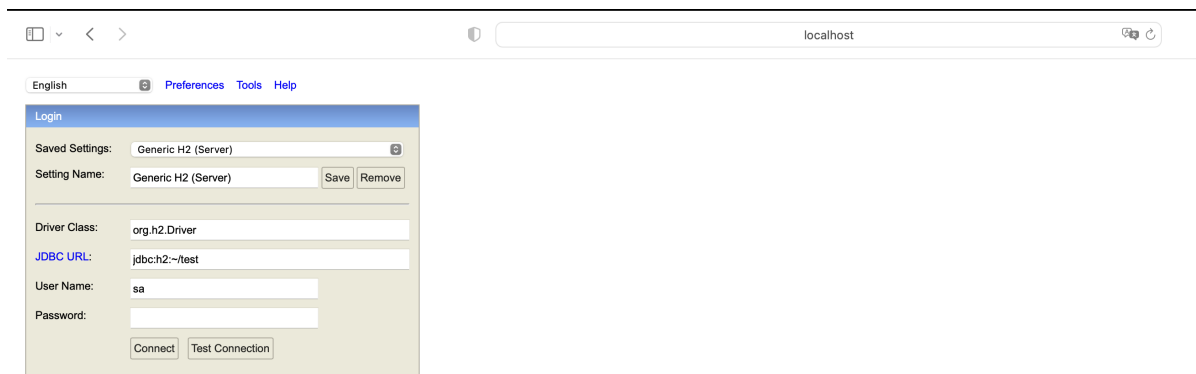
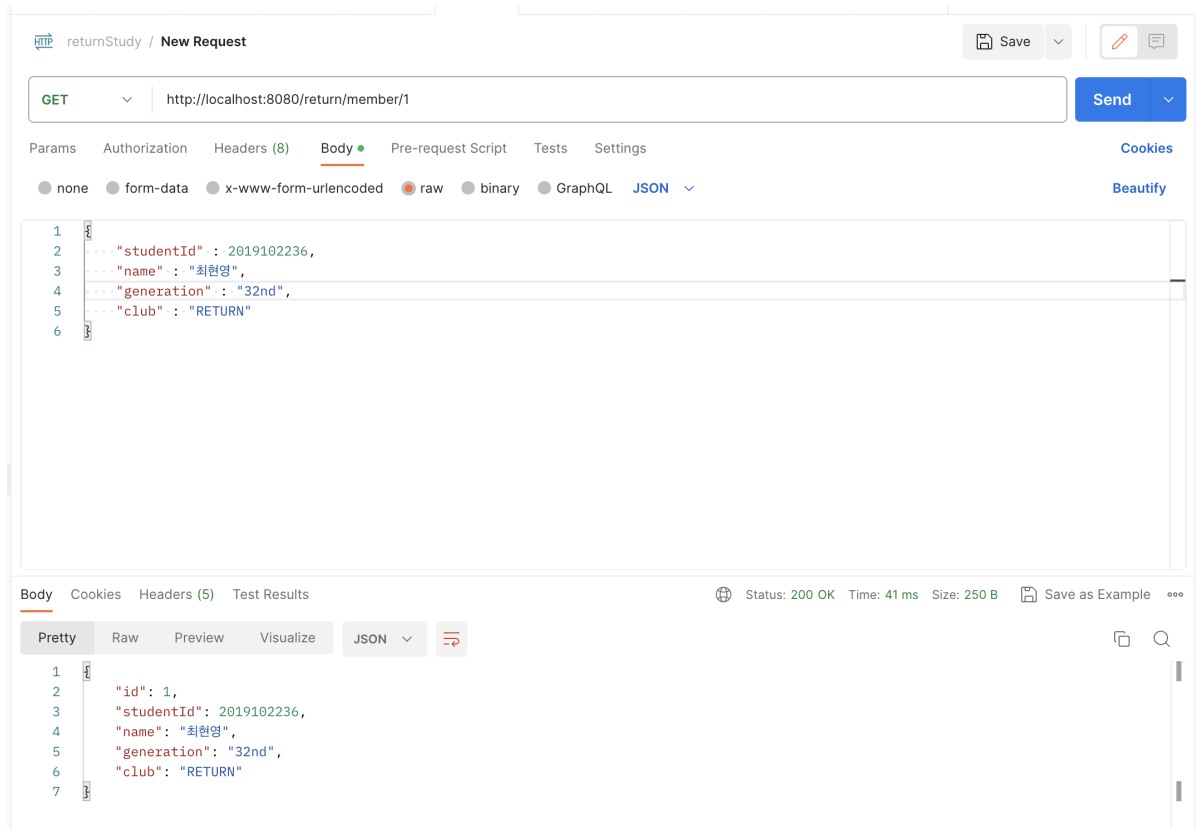
```

9. 지금까지 애플리케이션의 구조



10. 실행

- POST



<http://localhost:8080/console> 로 접속하면 다음과 같은 H2 접속 브라우저가 나온다. 다음과 같이 세팅하고 connect해주자.

Run
Run Selected
Auto complete
Clear
SQL statement:

select * from member;

```
select * from member;
```

STUDENT_ID	ID	CLUB	GENERATION	NAME
2019102236	1	RETURN	32nd	최현영

(1 row, 5 ms)

select 문으로 쿼리를 실행하면 방금 요청 바디에 담았던 데이터들이 데이터베이스에 정상적으로 저장되었습니다.

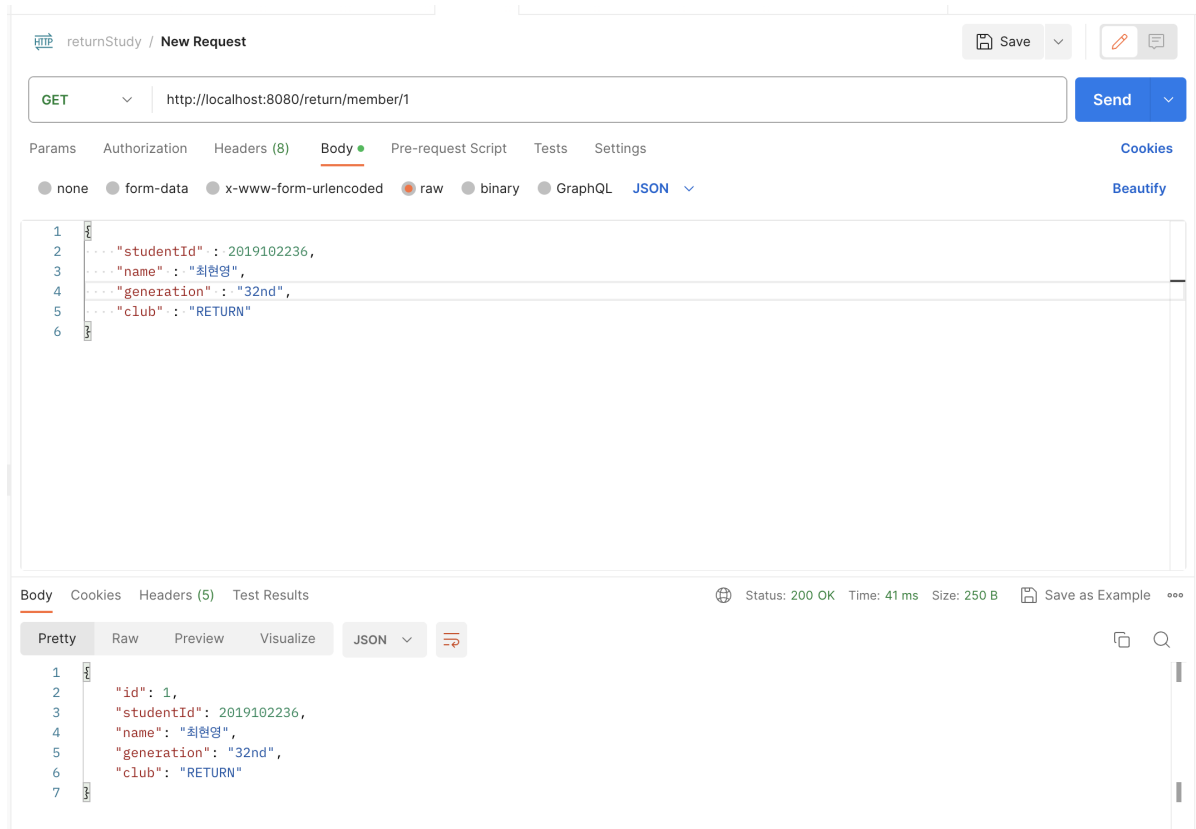
```

Hibernate:
    select
        next value for member_seq
Hibernate:
    insert
    into
        member
        (club,generation,name,student_id,id)
    values
        (?, ?, ?, ?, ?)

```

추가로 앞서 application.yml에서 sql 문을 표현하라고 설정하였다. 자동으로 JPA가 SQL을 작성하여 데이터베이스에 쿼리를 날리는 모습을 볼 수 있다. [SQL 문을 공부할 때 좋다]

- GET




방금 데이터베이스에 저장하고 반환된 ID[pk]를 PathParam으로 담아서 조회를 해보면 기존에 저장되어 있던 회원 정보가 반환된다.

- PUT



The screenshot shows a REST client interface for a PUT request. The URL is `http://localhost:8080/return/update/1`. The request body is a JSON object: `{ "studentId": 2019102236, "name": "최현영", "generation": "32nd", "club": "Aws Student Club" }`. The response status is `202 Accepted` with a response body: `{ "id": 1, "studentId": 2019102236, "name": "최현영", "generation": "32nd", "club": "Aws Student Club" }`.

ID값을 PathParam에 담아서 기존의 해당 엔티티를 가져온 뒤, 영속성 컨텍스트에 의해 더티체킹을 한 뒤, 변경된 필드만 새로 갱신한다. 최종적으로 변경된 회원 엔티티의 정보들을 반환하는 모습을 살펴볼 수 있다.

- DELETE

 returnStudy / **New Request Copy**

Save

DELETE

http://localhost:8080/return/withdraw/1

Send

Params

Authorization

Headers (8)

Body

Pre-request Script

Tests

Settings

Cookies

Query Params

	Key	Value	Description	...	Bulk Edit
	Key	Value	Description		

Body

Cookies

Headers (4)

Test Results

Status: 200 OK

Time: 40 ms

Size: 123 B

Save as Example

...

Pretty

Raw

Preview

Visualize

Text

1

ID값을 PathParam에 담아서 기존의 엔티티를 조회한 다음, 삭제한다.