2주차 키워드 정리

HTTP Method의 특징 (idempotent, Cacheability, Safe)

Method	멱등성(Idempotent)	Cacheability	Safe
GET	O	O	O
POST	X	△	X
PUT	O	X	X
PATCH	X	△	X
DELETE	O	X	X

(△ : 가능하지만 실제로 구현이 쉽지 않음)

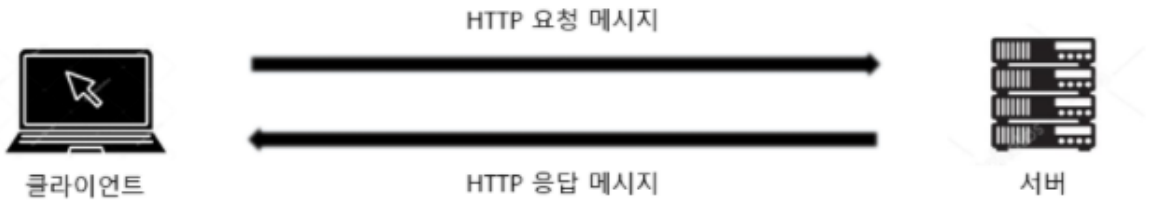
- **Idempotent**
  - 한 번 호출할 때와 여러 번 호출할 때의 결과가 같다.
  - f(f(x)) = f(x)
  - GET (O) : 호출 수에 관계없이 같은 결과가 조회된다.
  - PUT (O): 결과를 대체하므로 최종 결과는 같다.
  - DELETE (O): 결과를 삭제하므로 리소스가 삭제된 결과는 같다.
  - POST (X): 요청이 여러 번 들어가면 여러 번 결과가 쌓인다.

    - PATCH (X): 멱등성을 보장하지 않을 수도 있다.
        - 참고 : https://inpa.tistory.com/entry/WEB-🌐-HTTP의-멱등성---안정성---캐시성-💯-완벽-이해하기

- **Cacheability**
    - 응답 결과 리소스를 캐싱해서 효율적으로 사용할 수 있는가에 대한 여부이다.
    - 실제 실무에서는 GET, HEAD 정도만 캐시로 사용한다.

- **Safe**
    - 여러 번 해당 메소드를 호출해도 리소스를 변경하지 않는다.
    - GET을 제외한 메소드들은 안전하지 않다.
    - 안전이라는 개념은 해당 리소스가 변하는지 아닌지만을 고려한다.

@ResponseBody 와 @RequestBody 어노테이션의 역할



@RequestBody : 클라이언트 → 서버 요청

@ResponseBody : 서버 → 클라이언트 응답

- **@RequestBody**
    - json 기반의 HTTP Body를 자바 객체로 변환
    - 아래와 같이 요청이 들어오면,

```
POST /api/post HTTP/1.1
```

```
{
    "id": 1,
    "name": "user1"
}
```

```
// 객체
public class Entity{
  private Long id;
    private String name;
    private String address;
}

// Controller
@PostMapping("/api/post")
public void requestTest(@RequestBody Entity entity) {
    System.out.println("id = " + entity.id);
    System.out.println("name = " + entity.name);
    System.out.println("address = " + entity.address);
}
```

- entity.id == 1
- entity.name == "user1"
- entity.address == null

이렇게 Body의 내용을 객체로 변환해준다.

- **@ResponseBody**
  - 자바 객체를 json 기반의 HTTP Body로 변환
  - @RestController를 사용하면 리턴 값에 자동으로 @ResponseBody의 기능을 포함한다.

Java Record

```
public record Person (
  String name,
  String address
) {}
```

- 객체 간에 변경 불가능한 데이터를 전달 시 Java14 이전에는 boilerplate field(동일한 프로세스를 반복해야 함, 자동으로 업데이트 되지 않음.)와 메서드가 포함된 클래스를 생성해야 했기 때문에 사소한 실수가 발생할 수 있고 의도가 혼동될 가능성이 컸다.
- Java14에서 preview로 나온 뒤, Java16에서 정식 기능으로 포함되었다.
- 레코드는 **필드 유형과 이름만 필요한 불변 데이터 클래스**이다.
- equals, hashcode, tostring 메서드와 private, final field, public constructor는 Java 컴파일러에 의해 생성된다.
- 레코드는 다른 클래스를 상속받을 수 없다.

OOP와 RDB 사이의 패러다임의 불일치

- DB는 철저히 데이터 중심으로 설계되어 있으며 객체와 DB는 목적이 다르다.
- 객체에는 있는 상속, 다형성 같은 개념이 DB에는 없다.

→ 객체(OOP)와 관계형 DB의 데이터 표현방식과 다루는 방법이 달라 일어나는 현상이다.

1. **상속**
    - 객체는 상속의 기능을 가지지만 DB는 상속의 기능이 없다.

```
abstract class Item {
    Long id;
    String name;
    int price;
}

class Album extends Item {
    String artist;
}

class Movie extends Item {
    String director;
    String actor;
}

class Book extends Item {
    String author;
    String isbn;
}
```

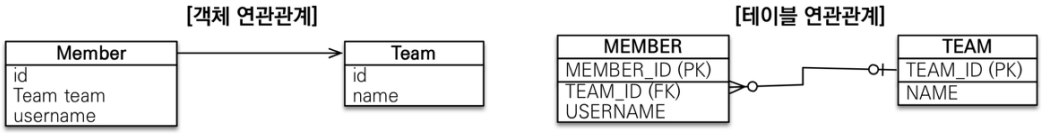
    만약 Album을 저장하려면 이 객체를 분해해서 여러 개의 SQL을 만들어야 한다.

INSERT INTO ITEM ...

INSERT INTO ALBUM ...

2. 연관관계

- 객체는 참조를 사용해 다른 객체와 연관관계를 가지고 참조에 접근해 연관된 객체를 조회한다.
- 반면, 테이블은 외래 키를 사용해 다른 테이블과 연관관계를 가지고 join을 사용해서 연관된 테이블을 조회한다.



- 이 경우, 객체의 참조 방향에서 문제가 생기게 된다.
  - 객체 연관관계의 경우 member.getTeam으로 참조가 가능하지만 Team.getMember()는 불가능하다.
  - 반면 테이블은 어느쪽에서든 join을 사용할 수 있다.

  - **객체를 테이블에 맞추어 모델링하는 경우**

```
class Member {
 String id; // MEMBER_ID 컬럼 사용
 Long teamId; // TEAM_ID FK컬럼 사용
 String username; // USERNAME 컬럼 사용
}

class Team {
 Long id; // TEAM_ID PK사용
 String name; // NAME 컬럼 사용
 }
```

  - TEAM_ID외래 키 까지 관계형 데이터베이스가 사용하는 방식에 맞추면 Member객체와 연관된 Team객체를 참조를 통해 조회할 수 없다.

- **객체지향 모델링**

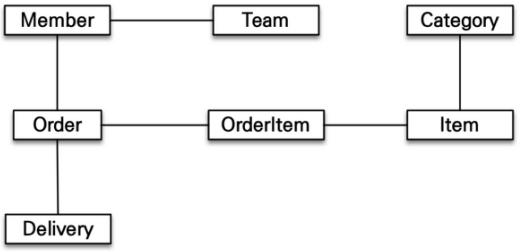
```
class Member {
 String id; // MEMBER_ID 컬럼 사용
 Team team; // 참조로 연관관계를 맺는다.
 String username; // USERNAME 컬럼 사용

 Team getTeam() {
   return team;
  }
}

class Team {
 Long id; // TEAM_ID PK사용
 String name; // NAME 컬럼 사용
 }
```

  - 객체지향 모델로 데이터를 저장하기 위해서는 참조 객체를 키값으로 변환해서 사용해야 한다.
    - member.getId()

3. **객체 그래프 탐색**



- DB에서는 객체를 조회할 때 member와 Team의 데이터만 조회할 경우 member.getOrder()은 null이 된다.
- SQL을 직접 다루면 처음 실행하는 SQL문에 따라 객체 그래프의 탐색이 한정된다.
- 객체는 객체 그래프를 마음대로 탐색할 수 있어야 하기 때문에 확신없이 탐색이 불가능하게 된다.
  - member.getOrder().getDelivery(); // 탐색이 가능할지 확신할 수 있는가?

4. 비교

- 데이터베이스는 기본 키의 값으로 각 로우를 구분한다.
- 객체는 동일성 비교(==)와 동등성 비교(equals())로 비교한다.
- 패러다임 불일치를 해결하기위해 같은 로우를 조회할 때마다 같은 인스턴스를 반환하도록 구현하는 것은 쉽지 않다.
- 같은 ID로 데이터를 조회했을 때 두 객체가 다르다. 같은 low에서 조회했지만 객체 측면에서 둘은 다른 인스턴스이다.

```
String memeberId = "100";
Member member1 = memberDAO.getMember(memberId);
Member member2 = memberDAO.getMember(memberId);

member1 == member2; // false
```

참고자료 : https://jgrammer.tistory.com/entry/JPA-패러다임-불일치?category=942980

@GeneratedValue의 옵션(생성 전략)

- JPA 엔티티에서 `@GeneratedValue` 어노테이션을 기본키 필드 위에 붙이면, 기본키(대체키)를 자동으로 생성해주는 전략을 사용할 수 있다.
  - IDENTITY 전략 (MYSQL용)
  - SEQUENCE 전략 (ORACLE용)
  - TABLE 전략 (모든 DB 사용)
  - AUTO 전략 (방언에 따라 자동으로 전략 지정)

- @GeneratedValue(strategy = GenerationType.**IDENTITY**)
  - 기본키 생성을 hibernate가 아니라, 데이터베이스가 하도록 위임한다.
  - MySQL, PostgreSQL에서 사용하는 기본키 자동생성 전략이다.
  - auto-increment를 사용하여 삽입된 레코드를 1씩 증가하는 방식으로 자동생성한다.

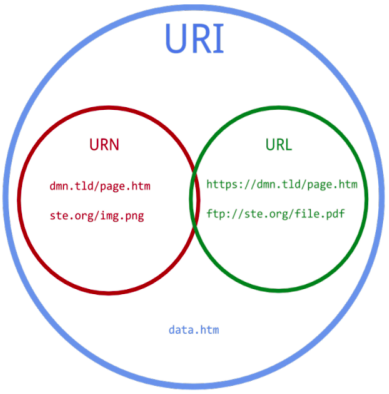
- @GeneratedValue(strategy = GenerationType.**SEQUENCE**)
  - 데이터베이스의 Sequence Object를 사용하여 데이터베이스가 자동으로 기본키를 생성해준다.

    - SEQUENCE 객체를 따로 생성하여 기본키를 생성한다.
    - ORACLE, PostgreSQL에서 사용하는 전략이다.

- @GeneratedValue(strategy = GenerationType.**TABLE**)
    - 키를 생성하는 테이블을 사용하는 방법으로 Sequence와 유사하다.

- @GeneratedValue(strategy = GenerationType.**AUTO**)
    - 기본 설정 값으로 각 데이터베이스에 따라 기본키를 자동으로 생성한다.

URL, URI



- **URL (Uniform Resource Locator)**
    - 리소스의 정확한 위치 정보(파일의 위치)를 나타낸다.
    - https://www.youtube.com/feed/subscriptions - 유튜브의 구독한 목록 URL

- **URI (Uniform Resource Identifier)**
    - 자원의 위치뿐만 아니라 자원에 대한 고유 식별자로서 URL을 의미를 포함한다.
    - 인터넷에서 요구되는 기본조건으로 인터넷 프로토콜에 항상 붙어다닌다.

- https://velog.io/@ziiyouth/category → URL, 카테고리는 리소스의 실제 위치

- https://velog.io/@ziiyouth/category/13 → URL을 포함한 URI, 자원의 경로 뒤에 식별자가 붙음

Path Parameter, Query String

- **Path parameter**
  - 원하는 조건의 데이터들 혹은 하나의 데이터에 대한 정보를 받아올 때에 적절하다.
  - 리소스를 식별해야하는 상황에서 더 적합하다.
  - 경로를 변수로 사용한다.
    - https://velog.io/@ziiyouth/**post/6**
- **Query string**
  - filtering, sorting, searching과 같은 상황에 적합하다.
  - 경로 뒤에 입력 데이터를 함께 제공하는 식으로 사용한다.
    - https://velog.io/@ziiyouth/post?post_id=6
