6월 7~8일

Optional

Optional이란?

Optional<T> 클래스를 사용하면 null 이 올 수 있는 값을 감싸는 Wrapper 클래스로, 참조하더라도 NPE(NullPointerException)가 발생하지 않도록 도와준다
Optional 클래스는 아래와 같이 value 값을 저장하기 때문에 값이 null이라도 NPE가 발생하지 않음

```
public final class Optional<T> {

  // If non-null, the value; if null, indicates no value is present
  private final T value;

  ...
}
```

for-each 문 : iterate 루프 돌릴 객체이고 iterate를 한개씩 순차적으로 var에 대입함

```
for (type var: iterate) {
    body-of-loop
}
```

인텔리 제이 단축키 : ^g = 다중 선택

[JPA] 기본기(PK) 매핑 방법 및 생성 전략

기본 키 매핑

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

1. 직접 할당
   - @Id 만 사용
2. 자동 생성
   - @Id와 @GeneratedValue를 같이 사용
   - 4가지 전략이 있음

자동 생성 전략
IDENTITY
개념

```
@GeneratedValue(strategy = GenerationType.IDENTITY)
```

기본 키 생성을 DB 에 위임
즉, id 값을 null 로 하면 DB 가 알아서 AUTO_INCREMENT 해준다
- Ex) MySQL, PostgreSQL, SQL Server DB2 등

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

```
// H2
create table Member (
  id varchar(255) generated by default as identity,
  ...
)
// MySQL
create table Member (
  id varchar(255) auto_increment,
  ...
)
```

특징
IDENTITY 전략은 entityManager.persist() 시점에 즉시 INSERT SQL을 실행하고 DB 에서 식별자를 조회한다.
- JPA는 보통 트랜잭션 commit 시점에 INSERT SQL을 실행한다

단점 : 모아서 INSERT 하는 것이 불가능하다.
- 하지만, 버퍼링해서 Write 하는 것이 큰 이득이 있지 않기 때문에 크게 신경쓰지 않아도 된다.
- 하나의 Transaction 안에서 여러 INSERT Query가 네트워크를 탄다고 해서 엄청나게 비약적인 차이가 나지 않는다.

SEQUENCE
개념

```
@GeneratedValue(strategy = GenerationType.SEQUNCE)
```

- DB Sequence Object를 사용
  - DB Sequence 는 유일한 값을 순서대로 생성하는 특별한 DB Object
  - 테이블 마다 Sequence Object를 따로 관리하고 싶으면 @SequenceGenerator에 sequenceName 속성을 추가한다.

- 즉, DB가 자동으로 숫자를 generate 해준다.
  EX) Oracle, PostgreSQL, DB2, H2 등

- @SequenceGenerator 필요

```
@Entity
@SequenceGenerator(
  name = "MEMBER_SEQ_GENERATOR",
  sequenceName = "MEMBER_SEQ", // 매핑할 데이터베이스 시퀀스 이름
  initialValue = 1,
  allocationSize = 1)
public class Member {
  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE,
                  generator = "MEMBER_SEQ_GENERATOR")
  private Long id;
}
```

```
// 1부터 시작해서 1씩 증가
create sequence MEMBER_SEQ start with 1 increment by 1
```

특징

- SEQUENCE 전략은 id 값을 설정하지 않고(null) generator에 매핑된 Sequence 전략(“MEMBER_SEQ”)에서 id 값을 얻어온다
  - 해당 Sequence Object는 DB가 관리하는 것이기 때문에 DB에서 id 값을 가져와야한다.

@SequenceGenerator 속성

속성	설명	기본값
name	식별자 생성기 이름	필수
sequenceName	데이터베이스에 등록되어 있는 시퀀스 이름	hibernate_sequence
initialValue	DDL 생성 시에만 사용됨, 시퀀스 DDL을 생성할 때 처음 1 시작하는 수를 지정한다.	1
allocationSize	시퀀스 한 번 호출에 증가하는 수 (성능 최적화에 사용), **데이터베이스 시퀀스 값이 하나씩 증가하도록 설정되어 있으면 이 값을 반드시 1로 설정해야 한다.**	50
catalog, schema	데이터베이스 catalog, schema 이름	

TABLE
개념

```
@GeneratedValue(strategy = GenerationType.TABLE)
```

- 키 생성 전용 테이블을 하나 만들어서 DB Sequence를 흉내내는 전략

- @TableGenerator 필요

```
@Entity
@SequenceGenerator(
  name = "MEMBER_SEQ_GENERATOR",
  table = "MY_SEQUENCES", // 데이터베이스 이름
  pkColumnValue = "MEMBER_SEQ",
  allocationSize = 1)
public class Member {
  @Id
  @GeneratedValue(strategy = GenerationType.TABLE,
                  generator = "MEMBER_SEQ_GENERATOR")
  private Long id;
}
```

```
create table MY_SEQUENCES (
   sequence_name varchar(255) not null,
   next_val bigint,
   primaty key ( sequence_name )
)
```

특징

- **장점 : 모든 DB에 적용가능**
- **단점 : 최적화 되어있지 않은 테이블을 직접 사용하기 때문에 성능상의 이슈가 있음**
- **운영 서버에서는 사용하기에 적합하지 않다.**
  - **DB에서 관례로 쓰는 것이 있기 때문에**

@TableGenerator 속성

속성	설명	기본값
name	식별자 생성기 이름	필수
table	키 생성 테이블 명	hibernate_sequences
pkColumnName	시퀀스 컬럼명	sequence_name
valueColumnNa	시퀀스 값 컬럼명	next_val
pkColumnValue	키로 사용할 값 이름	엔티티 이름
initialValue	초기 값, 마지막으로 생성된 값이 기준	0
allocationSize	시퀀스 한 번 호출에 증가하는 수 (성능 최적화에 사용)	50
catalog, schema	데이터베이스 catalog, schema 이름	
uniqueConstraints(DDL)	유니크 제약 조건을 지정할 수 있다.	

AUTO
개념

```
@GeneratedValue(strategy = GenerationType.AUTO)
```

- 기본 설정 값
- 상황에 따라 위 세 가지 전략을 자동으로 지정함

권장하는 식별자 전략
기본 키 제약 조건

1. null이 아니다.
2. 유일하다.
3. 변하면 안된다.

- 미래까지 이 조건을 만족하는 자연키는 찾기 어렵다. 그 대신 대리키/대체키를 사용하지.
  - 자연키 (Natural Key)
    비지니스적으로 의미가 있는 키 (Ex. 전화번호, 주민등록번호 등)
  - 대리키/대체키 (Generate Value)
    비지니스적으로 상관 없는 키 (Ex. Generate Value, 랜덤 값, 유휴 값 등)

권장하는 식별자 구성 전략

`(Long형) + (대체키) + (적절한 키 생성 전략) 사용`

1. Long Type
2. 대체키 사용
3. AUTO_INCREMENT 또는 Sequence Object 사용

JPA Cascade Types

JPA Cascade Type

- ALL
- PERSIST
- MERGE
- REMOVE
- REFRESH
- DETACH

CascadeType.ALL

- 상위 Entity에서 하위 Entity로 모든 작업을 전파

```java
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    private String name;

    @OneToMany(mappedBy = "person", cascade = CascadeType.ALL)
    private List<Address> addresses;
}
```

```java
@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;

    private String street;
    private int houseNumber;
    private String city;
    private int zipCode;

    @ManyToOne(fetch = FetchType.LAZY)
    private Person person;
}
```

CascadeType.PERSIST

- 하위 Entity까지 영속성 전달
  - Person Entity를 저장하면 Address Entity도 저장됨

```java
@Test
public void whenParentSavedThenChildSaved() {
    Person person = new Person();
    Address address = new Address();
    address.setPerson(person);
    person.setAddresses(Arrays.asList(address));
    session.persist(person); // persist 동작 수행
    session.flush();
    session.clear();
}
```

```
Hibernate: insert into Person (name, id) values (?, ?)
Hibernate: insert into Address (
    city, houseNumber, person_id, street, zipCode, id) values (?, ?, ?, ?, ?, ?)
```

CascadeType.MERGE
하위 Entity까지 병합 작업을 지속
- Address와 Person Entity를 조회한 후 업데이트

```java
@Test
public void whenParentSavedThenMerged() {
    int addressId;
    Person person = buildPerson("devender");
    Address address = buildAddress(person);
    person.setAddresses(Arrays.asList(address));
    session.persist(person);
    session.flush();
    addressId = address.getId();
    session.clear();

    Address savedAddressEntity = session.find(Address.class, addressId);
    Person savedPersonEntity = savedAddressEntity.getPerson();
    savedPersonEntity.setName("devender kumar");
    savedAddressEntity.setHouseNumber(24);
    session.merge(savedPersonEntity); // merge 동작 수행
    session.flush();
}
```

```
Hibernate: select address0_.id as id1_0_0_, address0_.city as city2_0_0_,
address0_.houseNumber as houseNum3_0_0_, address0_.person_id as person_i6_0_0_
, address0_.street as street4_0_0_, address0_.zipCode as zipCode5_0_0_ from Ad
dress address0_ where address0_.id=?

Hibernate: select person0_.id as id1_1_0_, person0_.name as name2_1_0_ from Pe
rson person0_ where person0_.id=?

Hibernate: update Address set city=?, houseNumber=?, person_id=?, street=?,
zipCode=? where id=?

Hibernate: update Person set name=? where id=?
```

CascadeType.REMOVE

- 하위 Entity 까지 제거 작업을 지속
  - 연결된 하위 Entity 까지 Entity 제거

```java
@Test
public void whenParentRemovedThenChildRemoved() {
    int personId;
    Person person = buildPerson("devender");
    Address address = buildAddress(person);
    person.setAddresses(Arrays.asList(address));
    session.persist(person);
    session.flush();
    personId = person.getId();
    session.clear();

    Person savedPersonEntity = session.find(Person.class, personId);
    session.remove(savedPersonEntity); // remove 동작 수행
    session.flush();
}
```

```
Hibernate: delete from Address where id=?
Hibernate: delete from Person where id=?
```

CascadeType.REFRESH

- DB로부터 인스턴스 값을 다시 읽어 오기(새로고침)
  - 연결된 하위 Entity까지 인스턴스 값 새로고침

```java
@Test
public void whenParentRefreshedThenChildRefreshed() {
    Person person = buildPerson("devender");
    Address address = buildAddress(person);
    person.setAddresses(Arrays.asList(address));
    session.persist(person);
    session.flush();
    person.setName("Devender Kumar");
    address.setHouseNumber(24);
    session.refresh(person); // refresh 동작 수행

    assertThat(person.getName()).isEqualTo("devender");
    assertThat(address.getHouseNumber()).isEqualTo(23);
}
```

CascadeType.DETACH
영속성 컨택스트에서 Entity 제거
- 연결된 하위 Entity까지 영속성 제거

```java
@Test
public void whenParentDetachedThenChildDetached() {
    Person person = buildPerson("devender");
    Address address = buildAddress(person);
    person.setAddresses(Arrays.asList(address));
    session.persist(person);
    session.flush();

    assertThat(session.contains(person)).isTrue();
    assertThat(session.contains(address)).isTrue();

    session.detach(person); // detach 동작 수행 시 영속성 컨텍스트에 존재하지 않음.
    assertThat(session.contains(person)).isFalse();
    assertThat(session.contains(address)).isFalse();
}
```

CS 스터디

38번 여러 작업을 수행하는 애플리케이션

- 애플리케이션

운영체제를 플랫폼으로 삼아 작업을 수행하는 온갖 종류의 프로그램이나 소프트웨어를 총칭하는 용어

애플리케이션은 하나의 특정 과제에 집중하거나 폭넓은 기능을 처리할 수도 있고 판매되거나 무료로 배포 할 수도 있음

애플리케이션의 코드는 소유권이 강하게 보호되거나 자유롭게 사용할 수 있는 오픈소스이거나 사용에 아무런 제한이 없음

애플리케이션의 크기는 천차만별이라 한가지 기능만 수행하는 독립적 프로그램부터 워드나

포토샵처럼 여러 가지 복잡합 작업을 수행하는 대형프로그램까지 다양하다

- 간단한 애플리케이션 예 : 날짜,시간등을 나타내는 date라는 유닉스 프로그램, 파일과 폴더를 나열하는 ls프로그램
- 복잡한 애플리케이션 예 : 워드, 포토샵등등

- 브라우저 : 규모가 크고 무료이고 오픈소스로 개발되는 애플리케이션
- 브라우저의 비동기적 이벤트 : 예측할 수 없는 시점에 일정한 순서를 따르지 않고 발생하는 이벤트

예시) 사용자가 링크를 클릭하면 브라우저는 페이지에 대한 요청을 보내는데, 브라우저는 해당 응답을 기다리고 있지 않고 사용자가 현재 페이지를 스크롤하면 즉각 반응하고 뒤로가기 버튼을 누르거나 다른 링크를 클릭하면 요청된페이지가 오는 중이라도 취소 하며 사용자가 창의 모양을 바꾸면 디스플레이를 계속 갱신해줘야함

- 브라우저는 정적인 텍스트부터 대화형 프로그램까지 많은 종류에 콘텐츠를 지원해야하는대 이중 일부를 확장프로그램에 콘텐츠 처리를 위임 할 수 있다
  PDF나 동영상같은 표준 포맷을 처리하는 데에 해당 방식을 사용하는것이 일반적

- 또한 브라우저는 수행한 작업이력, 북마크, 즐겨찾기 등의 다른 데이터를 유지하고 업로드, 다운로드, 이미지캐싱을 하기 위해 로컬 파일 시스템에 접근하기도 함

- 브라우저는 여러 수준에서 기능을 확장하기 위한 플랫폼 제공
  어도비 플래시, 자바스크립트용 가상머신, 애드블럭 플러스 등

- 위 내용들과 같은 모든 기능을 수행하기 위한 복잡한 코드를 포함하고 있기 때문에, 자체 구현 코드나 자신이 활성화하는 프로그램에 있는 버그를 이용하는 공격에 취약하다
  또 사용자의 순진함, 무지함, 무분별한 행동도 브라우저가 공격 받는대 한몫 함

- 자원을 관리하고, 동시에 일어나는 활동을 제어,조정하고 또한 다수의 출처에 정보를 저장하고 가져오며 애플리케이션 프로그램이 실행 될 수 있는 플랫폼을 제공하는 부분들을 보면 브라우저는 운영체제와 비슷하다.

- 그래서 한동안 브라우저를 운영체제로 사용하는 것이 가능해 보였고 그래서 시도해봤지만 10~20년 전까진 현실적인 장애물이 너무 많아 실행되진 못했다

- 하지만 오늘날 실행 가능한 대안이 됐고 수많은 서비스 들이 브라우저 인터페이스 만으로 접근할 수 있게됐다

39번 소프트웨어의 계층 구조