**CHATER1**

* 1. **SQL을 직접 다룰 때 발생하는 문제점**

**1.1.1 반복, 반복 그리고 반복**

-데이터베이스는 객체 구조와는 다른 데이터 중심의 구조를 가지므로 객체를 데이터베이스에 직접 저장허간 조회할 수 없다.

-중간에서 SQL과 JDBC API를 사용해서 변환 작업

🡺너무 많은 SQL과 JDBC API를 코드로 작성해야 된다.

**1.1.2 SQL에 의존적인 개발**

-필드를 추가하고, SQL 쿼리도 수정해야 된다.

-만약 자바 컬렉션이라면

ex)

list.add(member) // 등록

Member member = list.get(xxx); // 조회

Member.setTel(“xxx”); // 수정

-Member나 Team처럼 비즈니스 요구사항을 모델링한 객체를 엔티티 라고 한다.

-애플리케이션에서 SQL을 직접 다룰 때 문제점

1) 진정한 의미의 계층 분할이 어렵다.

2) 엔티티를 신뢰할 수 없다. (계속 변경)

3) SQL에 의존적인 개발을 피하기 어렵다.

**1.1.3 JPA와 문제 해결**

-JPA를 사용하면 SQL을 직접 작성하는 것이 아니라 JPA가 제공하는 API를 사용하면 된다.

그러면 JPA가 개발자 대신에 적절한 SQL을 생성해서 데이터베이스에 전달한다.

**1.2 패러다임의 불일치**

-객체지향 프로그래밍은 추상화, 캡슐화, 정보은닉, 상속, 다형성 등 복잡성을 제어할 수 있는 다양한 장치들을 제공한다.

-도메인 모델도 객체로 모델링하면 객체지향 언어가 가진 장점들을 활용할 수 있다.

-하지만 부모 객체를 상속받았거나, 다른 객체를 참조하고 있다면 객체의 상태를 저장하기 쉽지 않다.

**🡺그러면 관계형은?** 데이터 중심으로 구조화되어 있기 때문에 객체지향에서 이야기하는 추상화, 상속, 다형성 같은 개념이 없다.

**1.2.1 상속**

-테이블은 상속이라는 기능이 없다. 그나마 슈퍼타입 서브타입 관계를 사용하면 유사한 형태로 테이블을 설계할 수 있다.

**JPA와 상속**

-JPA는 상속과 관련된 패러다임의 불일치 문제를 개발자 대신 해결해준다. 개발자는 컬렉션에 객체를 저장하듯이JPA에게 객체를 저장하면 된다.

ex)

**저 장**

jpa.persist(album);

🡺INSERT INTO ITEM …

INSERT INTO ALBUM …

**조 회**

Album album = jpa.find(Album.class, albumId);

🡺 SELECT I.\*, A.\*

FROM ITEM i

JOIN ALBUM A ON I.ITEM\_ID = A.ITEM\_ID

**1.2.2 연관관계**

-객체는 참조를 사용해서 다른 객체와 연관관계를 가지고 참조에 접근해서 연관된 객체를 조회한다.

-반면에 테이블은 외래 키를 사용해서 다른 테이블과 연관관계를 가지고 조인을 사용해서 연관된 테이블을 조회한다.

-객체는 참조가 있는 방향으로만 조회할 수 있는 반면, 테이블은 외래 키 하나로 반대로도 조회 가능하다.

**잘못된 방법**

class Member {  
 String id; // MEMBER\_ID 컬럼 사용  
 Long teamId; // TEAM\_ID FK 컬럼 사용  
 String username; // USERNAME 컬럼 사용  
}  
  
class Team {  
 Long id; // TEAM\_ID PK 사용  
 String name; // NAME 컬럼 사용  
}

🡺Member 객체와 연관된 Team 객체를 참조를 통해서 조회할 수 없다.

**정상적인 방법**

class Member {  
 String id; // MEMBER\_ID 컬럼 사용  
 Team team; // 참조로 연관관계를 맺는다.  
 String username; // USERNAME 컬럼 사용  
}  
  
class Team {  
 Long id; // TEAM\_ID PK 사용  
 String name; // NAME 컬럼 사용  
}

🡺객체를 테이블에 저장하거나 조회하기 쉽지 않다. (테이블은 외래키가 있어야 된다.)

**JPA와 연관관계**

-JPA는 연관관계와 관련된 패러다임의 불일치 문제를 해결해준다.

🡺member.setTeam(team); // 회원과 팀 연관관계 설정

jpa.persist(member); // 회원과 연관관계 함께 저장

**1.2.3 객체 그래프 탐색**

-참조를 사용해서 연관된 객체를 찾는 것

-SQL을 직접 다루면 처음 실행하는 SQL에 따라 객체 그래프를 어디까지 탐색할 수 있는지 정해진다.

-객체 그래프 탐색은 코드만 보고 어디까지 할 수 있는지 예측할 수 없다. DAO를 열어서 SQL을 직접 확인해야 한다. (엔티티가 SQL 종속)

🡺연관된 모든 객체 그래프를 조회하면? 메모리 부담

**JPA와 객체 그래프 탐색**

-JPA를 사용하면 **지연 로딩**을 사용해서 신뢰된 객체를 마음껏 조회 가능

Order order = member.getOrder();  
order.getOrderDate(); // Order를 사용하는 시점에 SELECT ORDER SQL

🡺실제 객체를 불러 올 때 자주 사용되나 여부에 따라 PreLoading, LazyLoading 전략을 취한다.

**1.2.4 비교**

-동일성 비교 (==) : 인스턴스의 주소 값 비교

-동등성 비교 (equals) : 객체 내부의 값 비교

-SQL 조회

String memberId = "100";  
Member member1 = memberDAO.getMember(memberId);  
Member member2 = memberDAO.getMember(memberId);  
  
member1 == member2 // 다르다

🡺컬렉션이라면? 같았을텐데… 패러다임의 불일치

-JPA 조회

String memberId = "100";  
Member member1 = jpa.find(Member.class, memberId);  
Member member2 = jpa.find(Member.class, memberId);  
  
member1 == member2 // 같다

🡺동일하다.

**1.3 JPA란 무엇인가?**

-JPA는 자바 진영의 ORM 기술 표준



ORM은?

-객체와 관계형 데이터베이스 매핑

-ORM은 단순히 SQL을 개발자 대신 생성해서 데이터베이스에 전달할 뿐만 아니라 다양한 패러다임의 불일치까지 해결해준다.

**1.3.1 JPA 소개**

-EJB3.0에서 하이버네이트를 기반으로 새로운 자바 ORM 기술 표준이 만들어졌는데 이것이 JPA다.

-간략 버전 특징

1) JPA 1.0 (2006년) : 초기 버전. 복합 키와 연관관계 기능 부족

2) JPA 2.0 (2009년) : 대부분의 ORM 기능 포함. JPA Criteria가 추가됨

3) JPA 3.0 (2013년) : SP, 컨버터, 엔티티 그래프 기능등이 추가

**1.3.2 왜 JPA를 사용해야 하는가?**

생산성

-자바 컬렉션에 객체를 저장하듯이 JPA에게 저장할 객체를 전달하면 된다.

-지루하고 반복적인 코드와 CRUD용 SQL을 개발자가 직접 작성하지 않아도 된다.

유지보수

-번거로운 부분을 JPA가 대신 처리해주므로 수정해야 할 코드가 줄어든다.

-페러다임의 불일치도 해결해주므로 유지보수하기 좋은 도메인 모델을 편리하게 설계가능

패러다임의 불일치 해결

-JPA는 상속, 연관관계, 객체 그래프 탐색, 비교하기와 같은 패러다임의 불일치 문제를 해결해준다.

성능

String memberId = "100";  
Member member1 = jpa.find(Member.class, memberId);  
Member member2 = jpa.find(Member.class, memberId);

-트랜잭션 안에서 두번째 조회 시, 첫번째 조회결과를 재사용한다.

데이터 접근 추상화와 벤더 독립성

-관계형 데이터 베이스는 벤더마다 사용법이 다른 경우가 많다.

🡺기술에 종속됨

-JPA는 추상화 계층을 제공해서 데이터베이스 기술에 종속되지 않도록 한다.

🡺연결 정보만 바꾸면 된다.

**CHAPTER2**

**2.3 라이브러리와 프로젝트 구조**

-JPA 구현체로 하이버네이트를 사용하기 위한 핵심 라이브러리

* hibernate-core : 하이버네이트 라이브러리
* hibernate-entitymanager : 하이버네이트가 JPA 구현체로 동작하도록 JPA 표준을 구현한 라이브러리
* hibernate-jpa-2.1-api : JPA 2.1 표준 API를 모아둔 라이브러리

**2.3.1 메이븐과 사용 라이브러리 관리**

-메이븐은 라이브러리 관리 도구로서 pom.xml을 사용한다.

<dependencies>  
 <!-- JPA, 하이버네이트 -->  
 <dependency>  
 <groupId>org.hibernate</groupId>  
 <artifactId>hibernate-entitymanager</artifactId>  
 <version>${hibernate.version}</version>  
 </dependency>  
 <!-- H2 데이터베이스 -->  
 <dependency>  
 <groupId>com.h2database</groupId>  
 <artifactId>h2</artifactId>  
 <version>${h2db.version}</version>  
 </dependency>  
</dependencies>

-hibernate-entitymanager : JPA 표준과 하이버네이트를 포함.

* hibernate-core.jar
* hibernate-jpa-2.1.-api.jar

-h2 데이터베이스 : h2 데이터베이스에 접속

**2.4 객체 매핑 시작**

회원 테이블

**CREATE TABLE** MEMBER (  
 ID **VARCHAR**(255) **NOT NULL**, --아이디(기본 키)  
 **NAME VARCHAR**(255), --이름  
 AGE **INTEGER NOT NULL**, --나이  
 **PRIMARY KEY** (ID)  
)

회원 클래스

@Entity  
@Table(name = "MEMBER")  
public class Member {  
  
 @Id  
 @Column(name = "ID")  
 private String id;  
  
 @Column(name = "NAME")  
 private String username;  
  
 private Integer age;  
  
 public String getId() {  
 return id;  
 }  
 public void setId(String id) {  
 this.id = id;  
 }  
 public String getUsername() {  
 return username;  
 }  
 public void setUsername(String username) {  
 this.username = username;  
 }  
 public Integer getAge() {  
 return age;  
 }  
 public void setAge(Integer age) {  
 this.age = age;  
 }  
}

-@Entity : 테이블과 매핑한다고 JPA에 알려준다. @Entity가 사용된 클래스를 엔티티 클래스라고 한다.

-@Table : 엔티티 클래스에 매핑할 테이블 정보를 알려준다. 생략하면 클래스 이름과 매핑

-@Id : 엔티티 클래스의 필드를 테이블의 기본 키(Primary key)에 매핑한다. (식별자 필드)

-@Column : 필드를 컬럼에 매핑한다.

-매핑 정보가 없는 필드 : 필드명을 사용해서 컬럼명으로 매핑한다. (대소문자 구분할 경우 명시적으로 @Column(name=”AGE”) )

**2.5 persistence.xml 설정**

META-INF/persistence.xml 클래스 패스 경로에 있으면 별도의 설정 없이 JPA가 인식한다.

(Spring에서는 root-context.xml에 설정)

<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence" version="2.1">  
 <persistence-unit name="jpabook">  
 <properties>  
 <!-- 필수 속성 -->  
 <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>  
 <property name="javax.persistence.jdbc.user" value="sa"/>  
 <property name="javax.persistence.jdbc.password" value=""/>  
 <property name="javax.persistence.jdbc.url" value="jdbc:h2:tcp://localhost/~/test"/>  
 <property name="hibernate.dialect" value="org.hibernate.dialect.H2Dialect" />  
  
 <!-- 옵션 -->  
 <property name="hibernate.show\_sql" value="true" />  
 <property name="hibernate.format\_sql" value="true" />  
 <property name="hibernate.use\_sql\_comments" value="true" />  
 <property name="hibernate.id.new\_generator\_mappings" value="true" />  
  
 <!--<property name="hibernate.hbm2ddl.auto" value="create" />-->  
 </properties>  
 </persistence-unit>  
</persistence>

JPA 표준 속성

-javax.persistence.jdbc.driver : JDBC 드라이버

-javax.persistence.jdbc.user : 데이터베이스 접속 아이디

-javax.persistence.jdbc.password : 데이터베이스 접속 비밀번호

-javax.persistence.jdbc.url : 데이터베이스 접속 URL

하이버네이트 속성

-hibernate.dialect: 데이터베이스 설정

**2.5.1 데이터베이스 방언**

-데이터베이스 방언 : 데이터베이스 고유 기능

🡺하이버네이트를 포함한 대부분 JPA 구현체는 이런 문제를 해결하고자 데이터베이스 방언 클래스를 제공한다.

<https://docs.jboss.org/hibernate/orm/4.3/manual/en-US/html_single/#configuration-optional-dialects>

<property name="hibernate.show\_sql" value="true" />  
<property name="hibernate.format\_sql" value="true" />  
<property name="hibernate.use\_sql\_comments" value="true" />  
<property name="hibernate.id.new\_generator\_mappings" value="true" />

-hibernate.show.sql : 하이너베이트가 실행한 SQL을 출력한다.

-hibernate.format\_sql : 하이버네이트가 실행한 SQL을 출력할 때 보기 쉽게 정렬한다.

-hibernate.use\_sql\_comments : 쿼리를 출력할 때 주석도 함께 출력한다.

-hibernate.id.new\_generator\_mappings : JPA 표준에 맞춘 새로운 키 생성 전략을 사용한다.

**2.6 애플리케이션 개발**

public static void main(String[] args) {  
  
 //엔티티 매니저 팩토리 생성  
 EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("jpabook");  
 EntityManager em = emf.createEntityManager(); //엔티티 매니저 생성  
 EntityTransaction tx = em.getTransaction(); //트랜잭션 기능 획득  
  
 try {  
 tx.begin(); //트랜잭션 시작  
 *logic*(em); //비즈니스 로직  
 tx.commit();//트랜잭션 커밋  
  
 } catch (Exception e) {  
 e.printStackTrace();  
 tx.rollback(); //트랜잭션 롤백  
 } finally {  
 em.close(); //엔티티 매니저 종료  
 }  
  
 emf.close(); //엔티티 매니저 팩토리 종료  
}

크게 3개로 나누어져 있다.

1) 엔티티 매니저 설정

2) 트랜잭션 관리

3) 비즈니스 로직

**2.6.1 엔티티 매니저 설정**

엔티티 매니저 팩토리 생성

EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("jpabook");

-persistence.xml의 설정 정보를 사용해서 엔티티 매니저 팩토리 생성

-엔티티 매니저 팩토리는 비용이 비싸므로 한 번만 생성하고 공유해서 사용한다.

엔티티 매니저 생성

EntityManager em = emf.createEntityManager();

-JPA의 기능 대부분은 엔티티 매니저가 제공

-엔티티를 데이터베이스에 등록/수정/삭제/조회할 수 있다.

-내부에 데이터소스를 유지하면서 데이터베이스와 통신한다. (애플리케이션 입장에서는 가상의 데이터베이스)

-데이터베이스 커넥션과 밀접한 관계가 있으므로 스레드간에 공유하거나 재사용하면 안 된다.

**2.6.2 트랜잭션 관리**

-JPA를 사용할 경우 항상 트랜잭션 안에서 데이터를 변경해야 한다.

EntityTransaction tx = em.getTransaction(); //트랜잭션 기능 획득  
  
try {  
 tx.begin(); //트랜잭션 시작  
 *logic*(em); //비즈니스 로직  
 tx.commit();//트랜잭션 커밋  
  
} catch (Exception e) {  
 e.printStackTrace();  
 tx.rollback(); //트랜잭션 롤백  
}

-로직이 정상 동작하면 commit하고 예외가 발생하면 rollback 한다.

**2.6.3 비즈니스 로직**

public static void logic(EntityManager em) {  
 //등록  
 em.persist(member);  
  
 //수정  
 member.setAge(20);  
  
 //한 건 조회  
 Member findMember = em.find(Member.class, id);  
  
 //목록 조회  
 List<Member> members = em.createQuery("select m from Member m", Member.class).getResultList();  
  
 //삭제  
 em.remove(member);

등록

-엔티티를 저장하려면 persist() 메소드에 저장할 엔티티를 넘겨주면 된다.

수정

-update문이 없고 도메인에서 set을 하면 트랜잭션 범위내에서 자동으로 저장된다.

삭제

-remove() 메소드를 이용해서 삭제

한 건 조회

-@Id로 데이터베이스 테이블의 기본 키와 매핑한 식별자 값으로 조회 한다.

**2.6.4 JPQL**

-JPA는 SQL을 추상화한 JPQL이라는 객체지향 쿼리 언어를 제공한다.

-SQL과 JPQL의 차이점

* JPQL은 엔티티 객체를 대상으로 쿼리한다. (클래스와 필드를 대상으로 쿼리)
* SQL은 데이터베이스 테이블을 대상으로 쿼리한다.

em.createQuery("select m from Member m", Member.class).getResultList();

-여기서 Member는 테이블이 아니라 엔티티 객체를 말하는것이다. JPQL은 데이터베이스 테이블을 전혀 알지 못한다.

🡺JPA가 JPQL을 분석 후, SQL을 만들어서 데이터베이스에서 데이터를 조회한다.

**CHAPTER3**

**3.1 엔티티 매니저 팩토리와 엔티티 매니저**

-엔티티 매니저는 데이터베이스 연결이 꼭 필요한 시점까지 커넥션을 얻지 않는다. (트랜잭션을 시작할 때 획득)

**3.2 영속성 컨텍스트란? (엔티티를 영구 저장하는 환경)**

-em.persist() : 엔티티 매니저를 사용해서 회원 엔티티를 영속성 컨텍스트에 저장한다.

-영속성 컨텍스트는 엔티티 매니저를 생성할 때 하나 만들어 진다.

**3.3 엔티티의 생명주기**

-엔티티에는 4가지 상태 존재

* 비영속(new/transient) : 영속성 컨텍스트와 전혀 관계가 없는 상태
* 영속(managed) : 영속성 컨텍스트에 저장된 상태
* 준영속(detached) : 영속성 컨텍스트에 저장되었다가 분리된 상태
* 삭제(removed) : 삭제된 상태

**Lifecycle**



**비영속**

-엔티티 객체를 생성했다. 순수한 객체 상태이며 아직 저장하지 않았다.

Member member = new Member();  
member.setId(id);  
member.setUsername("지한");

**영속**

-엔티티 매니저를 통해서 엔티티를 영속성 컨텍스트에 저장했다.

-영속성 컨텍스트가 관리하는 엔티티를 영속 상태라고 한다.

em.persist(member);

**준영속**

-영속 상태의 엔티티를 영속성 컨텍스트가 관리하지 않으면 준영속 상태가 된다.

em.detach(member);

**삭제**

-엔티티를 영속성 컨텍스트와 데이터베이스에서 삭제한다.

em.remove(member);

**3.4. 영속성 컨텍스트의 특징**

**영속성 컨텍스트와 식별자 값**

-영속 상태는 식별자 값이 반드시 있어야 한다.

**영속성 컨텍스트와 데이터베이스 저장**

-트랜잭션을 커밋하는 순간 데이터베이스에 반영하는데 이것을 플러시(flush)라 한다.

**영속성 컨텍스트가 엔티티를 관리시 장점**

-1차캐시, 동일성 보장, 쓰기 지연, 변경 감지, 지연 로딩 등

**3.4.1 엔티티 조회**

-영속성 컨텍스트는 내부에 캐시를 가지고 있는데(1차 캐시) 영속 상태의 엔티티는 모두 이곳에 저장

-em.find()를 호출 하면 먼저 1차 캐시에서 엔티티를 찾고 없으면 데이터베이스 조회

**1차 캐시에서 조회**

Member member = new Member();  
member.setId(id);  
member.setUsername("지한");

// 1차 캐시에 저장됨  
em.persist(member);  
  
// 1차 캐시에서 조회

Member findMember = em.find(Member.class, id);

**데이터베이스에서 조회**

-1차 캐시에 없으면 엔티티 매니저는 데이터베이스를 조회. 그리고 1차 캐시에 저장한 후에 영속 상태의 엔티티를 반환. (성능상 이점)

**영속 엔티티의 동일성 보장**

String memberId = "100";  
Member member1 = jpa.find(Member.class, memberId);  
Member member2 = jpa.find(Member.class, memberId);  
  
member1 == member2 // 같다

**3.4.2 엔티티 등록**

-트랜잭션을 커밋하기 직전까지 데이터베이스에 엔티티를 저장하지 않고 내부 쿼리 저장소

(쓰기 지연 SQL 저장소)에 데이터를 모아 놓는다. 그리고 트랜잭션을 커밋할 때 모아둔 쿼리를 데이터베이스로 보낸다.

🡺트랜잭션을 지원하는 **쓰기 지연** 이라고 한다.

-트랜잭션 커밋하면 쓰기 지연 SQL 저장소에 모인 쿼리를 데이터베이스에 보낸다.

**3.4.3 엔티티 수정**

**SQL 수정 쿼리의 문제점**

-수정 쿼리가 많아지는 것은 물론이고 비즈니스 로직을 분석하기 위해 SQL을 계속 확인해야 한다.

**변경감지 (JPA는?)**

transaction.begin(); //트랜잭션 시작  
  
// 영속 엔티티 조회  
Member memberA = em.find(Member.class, "memberA");  
  
// 영속 엔티티 데이터 수정  
memberA.setUsername("hi");  
memberA.setAge(10);  
  
transaction.commit();//트랜잭션 커밋

-단순히 엔티티를 조회해서 데이터만 변경하면 된다.

-이렇게 엔티티의 변경사항을 데이터베이스에 자동으로 반영하는 기능을 **변경감지** 라 한다.

-JPA는 엔티티를 영속성 컨텍스트에 보관할 때, 최초 상태를 복사해서 저장해두는데 이것을 스냅샷이라고 한다. 그리고 플러시 시점에 스냅샷과 엔티티를 비교해서 변경된 엔티티를 찾는다.

-변경 감지는 영속 상태의 엔티티에만 적용된다.

**JPA의 기본 전략은 엔티티의 모든 필드를 업데이트한다. (장점)**

-모든 필드를 사용하면 수정 쿼리가 항상 같다. (재사용)

-데이터베이스에 동일 쿼리를 보내면 이전에 파싱된 쿼리를 재사용할 수 있다.

-필드가 많거나 저장되는 내용이 크면 수정된 데이터만 사용해서 동적으로 Update 하면 된다.

(@DynamicUpdate)

@Entity  
@DynamicUpdate  
@Table(name = "MEMBER")  
public class Member {

. . .

}

**3.4.4 엔티티 삭제**

-em.remove를 호출하는 순간 영속성 컨텍스트에서 제거된다.

**3.5 플러시**

-플러시(flush)는 영속성 컨텍스트의 변경 내용을 데이터베이스에 반영한다.

-영속성 컨텍스트는 비우지 않는다.

**플러시 하는 3가지 방법**

1) em.flush() 직접 호출

2) 트랜잭션 커밋 시 플러시가 자동 호출

3) JPQL 쿼리 실행 시 플러시가 자동 호출

**직접 호출**

-flush() 메소드를 직접 호출해서 강제로 플러시. 거의 테스트때만 사용

(로직상 먼저 commit 되어야 할 때도 필요할 것 같음)

**트랜잭션 커밋 시 플러시 자동 호출**

-트랜잭션을 커밋하기 전에 꼭 플러시를 호출해서 영속성 컨텍스트의 변경 내용을 데이터베이스에 반영해야 한다. (JPA는 자동으로 호출)

**JPQL 쿼리 실행 시 플러시 자동 호출**

em.persist(member);  
List<Member> members = em.createQuery("select m from Member m",

-empersist는 영속성 컨텍스트에 있지만 아직 데이터베이스에 반영되지 않았다. 이때 JPQL을 실행하면 쿼리 결과가 조회되지 않기 때문에 (JPQL은 SQL로 변환되어 데이터베이스 조회)

자동으로 플러시를 호출 후, 조회 한다.

**3.5.1 플러시 모드 옵션**

-FlushModeType.AUTO : 커밋이나 쿼리를 실행할 때 플러시 (default)

-FlushModeType.COMMIT : 커밋할 때만 플러시

em.setFlushMode(FlushModeType.*COMMIT*);

**3.6. 준영속**

-준영속 상태의 엔티티는 영속성 컨텍스트가 제공하는 기능을 사용할 수 없다.

**준영속 상태로 만드는 3가지 방법**

1) em.detach : 특정 엔티티만 준영속 상태로 전환

2) em.clear() : 영속성 컨텍스트를 완전히 초기화 한다.

3) em.close() : 영속성 컨텍스트를 종료한다.

**3.6.1 엔티티를 준영속 상태로 전환 : detach()**

public void testDetached() {  
 EntityManagerFactory emf = Persistence.*createEntityManagerFactory*("jpabook");  
 EntityManager em = emf.createEntityManager(); //엔티티 매니저 생성  
 EntityTransaction transaction = em.getTransaction();  
  
 Member member = new Member();  
 member.setId("memeberA");  
 member.setUsername("회원A");  
  
 // 회원 엔티티 영속 상태  
 em.persist(member);  
  
 // 회원 엔티티를 영속성 컨텍스트에서 분리, 준영속 상태  
 em.detach(member);  
 transaction.commit();  
}

-detach 호출 순간 1차 캐시부터 쓰기 지연 SQL 저장소까지 해당 엔티티를 관리하기 위한 모든 정보가 제거된다.

-쓰기 지연 SQL 저장소의 INSERT SQL도 제거되어서 데이터베이스에 저장되지도 않는다.

**3.6.2 영속성 컨텍스트 초기화 : clear()**

-em.detach()가 특정 엔티티 하나를 준영속으로 만들었다면 em.clear()는 모든 엔티티를 준영속 상태로 만든다.

Member member = em.find(Member.class, "memberA");  
em.clear();

member.setUsername(“changeName”);

-영속성 컨텍스트를 제거하고 새로 만든 것과 같다.

-당연히 set을 해도 데이터베이스에 반영되지 않는다.

**3.6.3 영속성 컨텍스트 종료 : close()**

-영속성 컨텍스트가 종료되어 관리되지 않는다. (준영속 상태)

**3.6.4 준영속 상태의 특징**

-거의 비영속 상태에 가깝다.

-식별자 값을 가지고 있다. (비영속과 차이)

-지연 로딩을 할 수 없다. (영속성 컨텍스트가 초기화)

**3.6.5 병합: merge()**

-준영속 상태의 엔티티를 다시 영속 상태로 변경하려면 병합을 사용한다. (새로운 영속 상태 반환)

-병합은 준영속, 비영속을 신경 쓰지 않는다. (save or update)

**CHAPTER4**

**JPA의 주요 어노테이션**

-객체와 테이블 매핑 : @Entity, @Table

-기본 키 매핑 : @Id

-필드와 컬럼 매핑 : @Column

-연관관계 매핑 : @manyToOne, @JoinColumn

**4.1 @Entity**

-테이블과 매핑할 클래스는 @Entity 어노테이션 필수

|  |  |  |
| --- | --- | --- |
| 속성 | 기능 | 기본값 |
| name | JPA에서 사용할 엔티티 이름 지정.  보통 기본값이 클래스 이름 | 클래스 이름 |

-기본 생성자 필수

🡺파라미터 생성자만 있는 경우 에러남

-final 클래스, enum, interface, inner 클래스에 사용 불가

-저장할 필드에 final 사용 금지

**4.2 @Table**

-엔티티와 매핑할 테이블을 지정한다. 생략 시 엔티티 이름을 테이블 이름으로 사용

|  |  |  |
| --- | --- | --- |
| 속성 | 기능 | 기본값 |
| name | 매핑할 테이블 이름 | 엔티티 이름을 사용 |
| catalog | Catalog 기능이 있는 데이터베이스에서 catalog를 매핑 |  |
| schema | Schema 기능이 있는 데이터베이스에서 schema를 매핑 |  |
| uniqueConstraints (DDL) | DDL 생성 시에 유니크 제약조건을 만든다.  (스키마 자동 생성 기능을 사용해서 DDL  만들때만 사용) |  |

**4.3 다양한 매핑 사용**

@Enumerated(EnumType.*STRING*)  
private RoleType type;  
  
@Temporal(TemporalType.*TIMESTAMP*)  
private Date createDate;  
  
@Temporal(TemporalType.*TIMESTAMP*)  
private Date lastModifiedDate;  
  
@Lob  
private String description;

-@Enumerated : 자바의 enum type 사용 시

-@Temporal : 자바의 날짜 타입을 사용 시

-@Lob : CLOB, BLOB 타입을 매핑할 수 있음

**4.4 데이터베이스 스키마 자동 생성**

persistence.xml

<property name="hibernate.hbm2ddl.auto" value="create" />

-애플리케이션 실행 시점에 데이터베이스 테이블을 자동으로 생성

<property name="hibernate.show\_sql" value="true" />

-해당 옵션 사용시 실행되는 DDL을 출력할 수 있음

**DDL 콘솔 출력**

Hibernate:  
 drop table MEMBER if exists  
Hibernate:  
 create table MEMBER (  
 ID varchar(255) not null,  
 age integer,  
 createDate timestamp,  
 description clob,  
 lastModifiedDate timestamp,  
 type varchar(255),  
 NAME varchar(255),  
 primary key (ID)  
 )

-기존 테이블을 삭제하고 다시 생성

-해당 DDL이 완벽하지 않으므로 개발 환경에서 사용하거나 매핑 참고용으로만 사용하는 것이 좋다.

**hibernate.hbm2ddl.auto 옵션**

|  |  |
| --- | --- |
| 옵션 | 설명 |
| create | 기존 테이블을 삭제하고 새로 생성 (DROP + CREATE) |
| create-drop | 애플리케이션을 종료할 때 생성한 DDL을 제거  (DROP + CREATE + DROP) |
| update | 데이터베이스 테이블과 엔티티 매핑정보를 비교해서 변경 사항만 수정 |
| validate | 데이터베이스 테이블과 엔티티 매핑정보를 비교해서 차이가 있으면 경고를 남기고 애플리케이션 미실행. |
| none | 자동 생성 기능을 사용하지 않는다. |