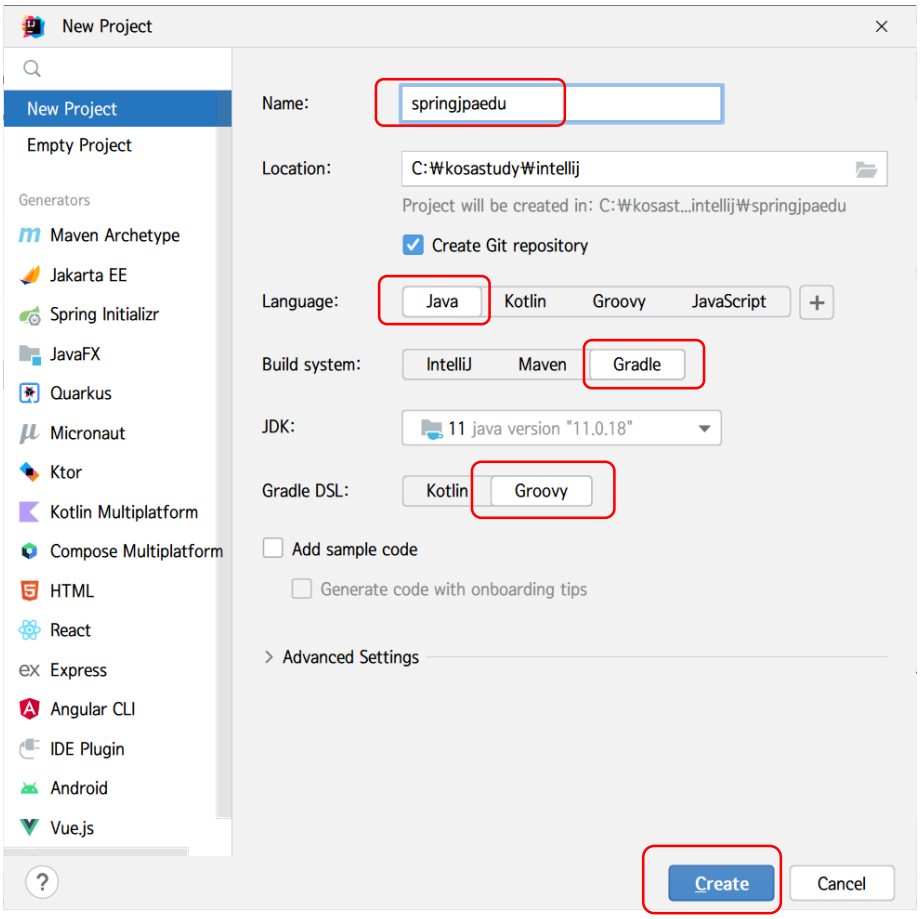
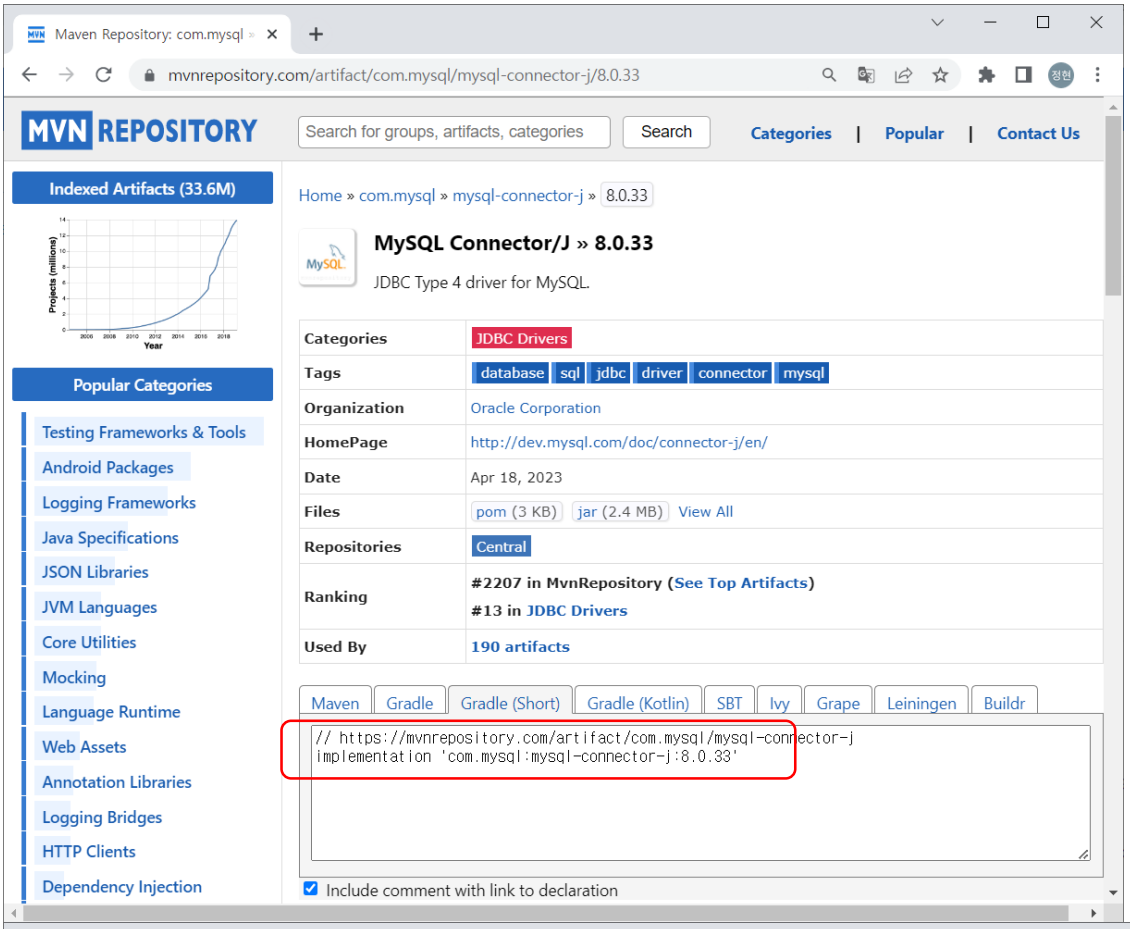
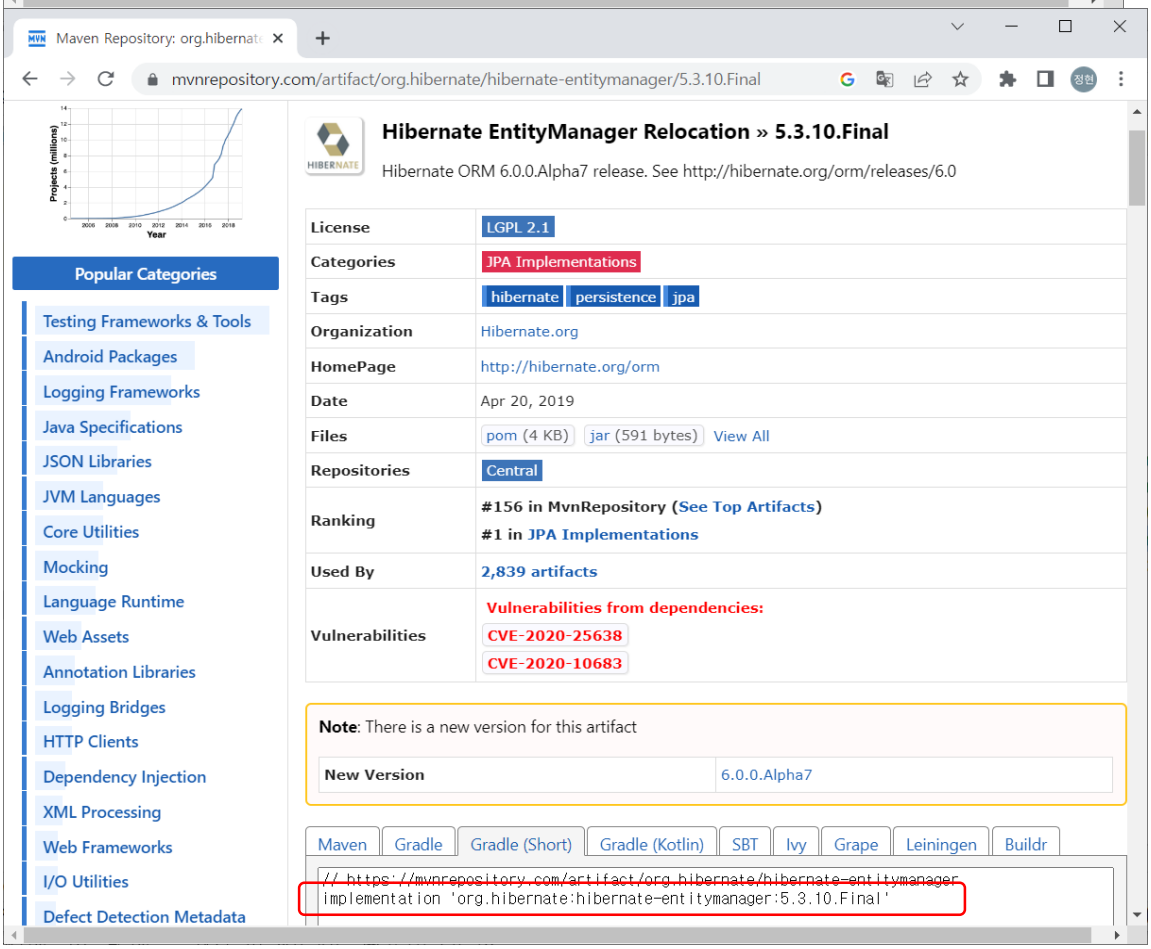
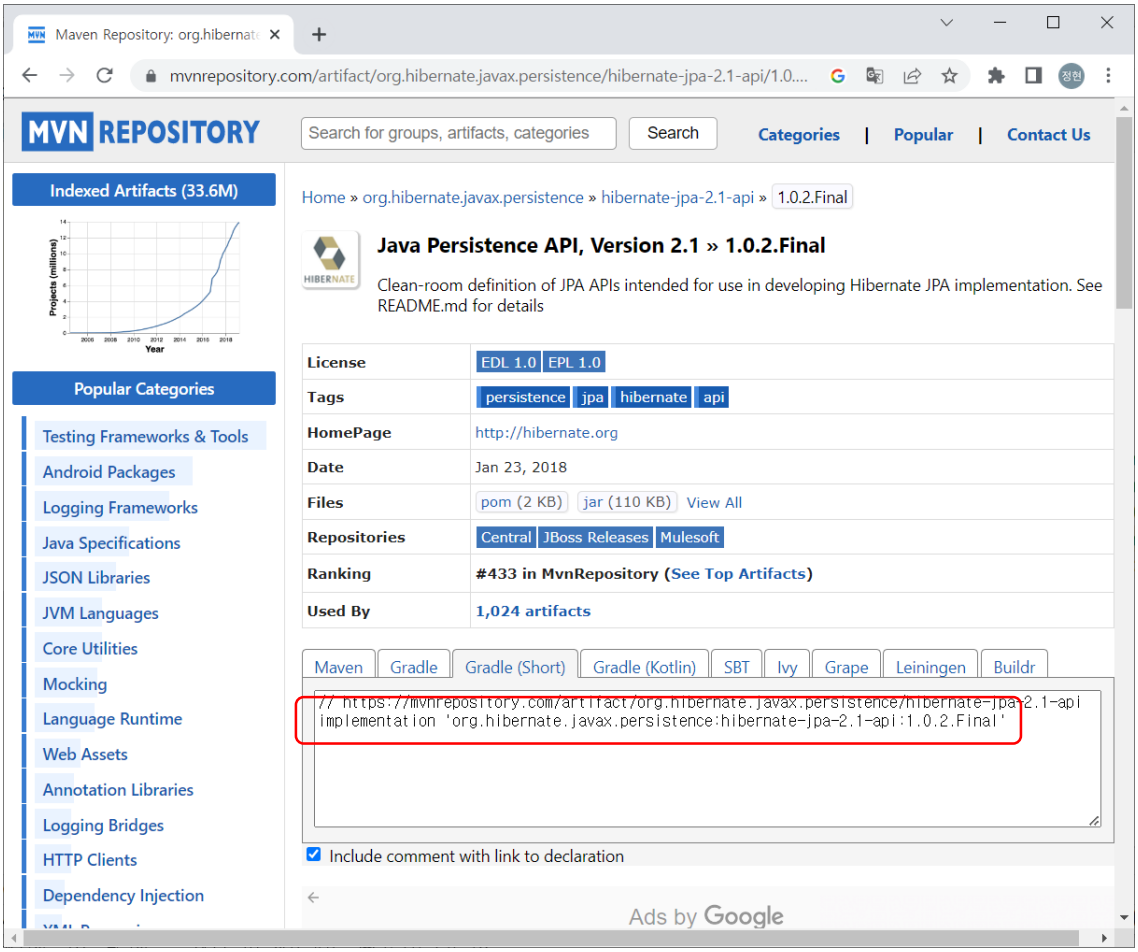
[JPA 테스트 환경 구축]

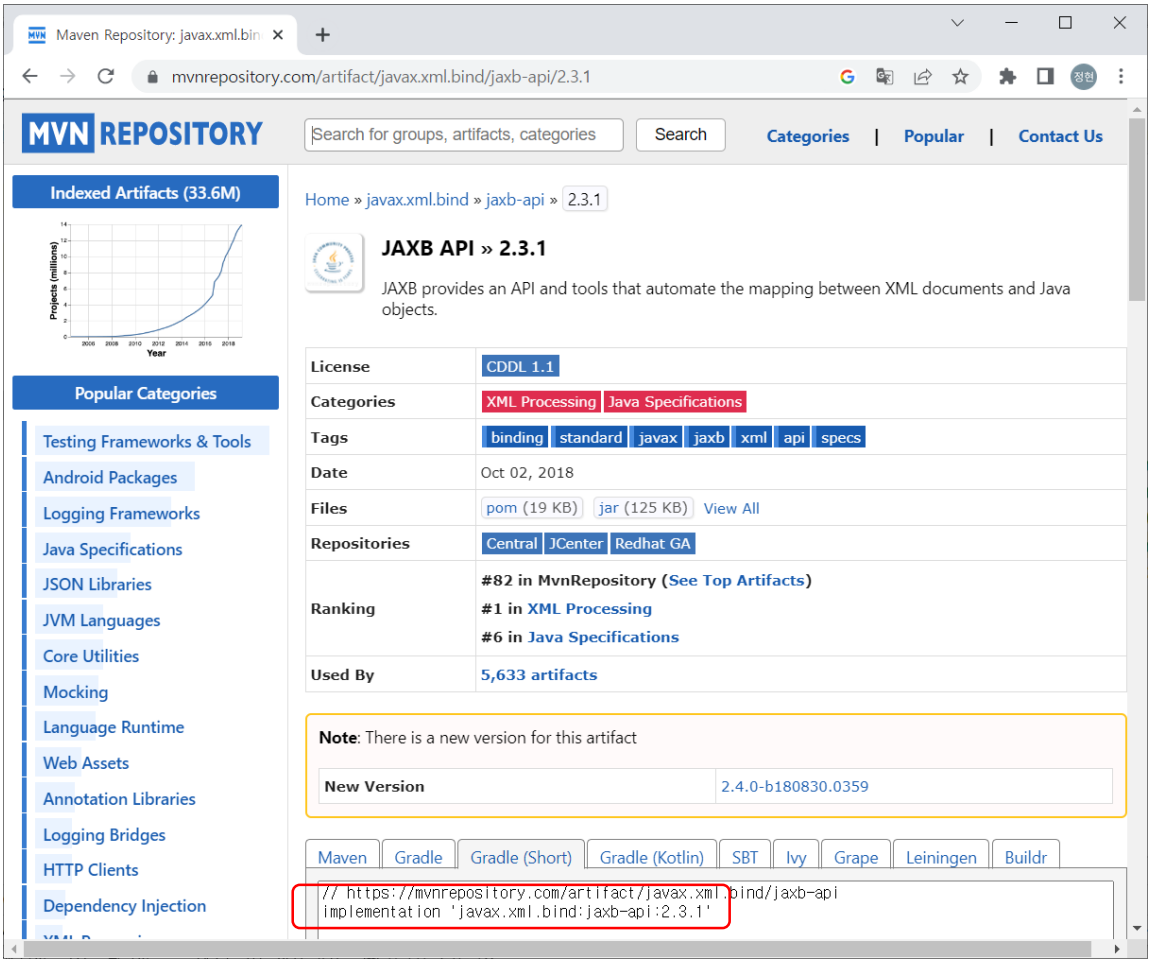
- 프로젝트 생성

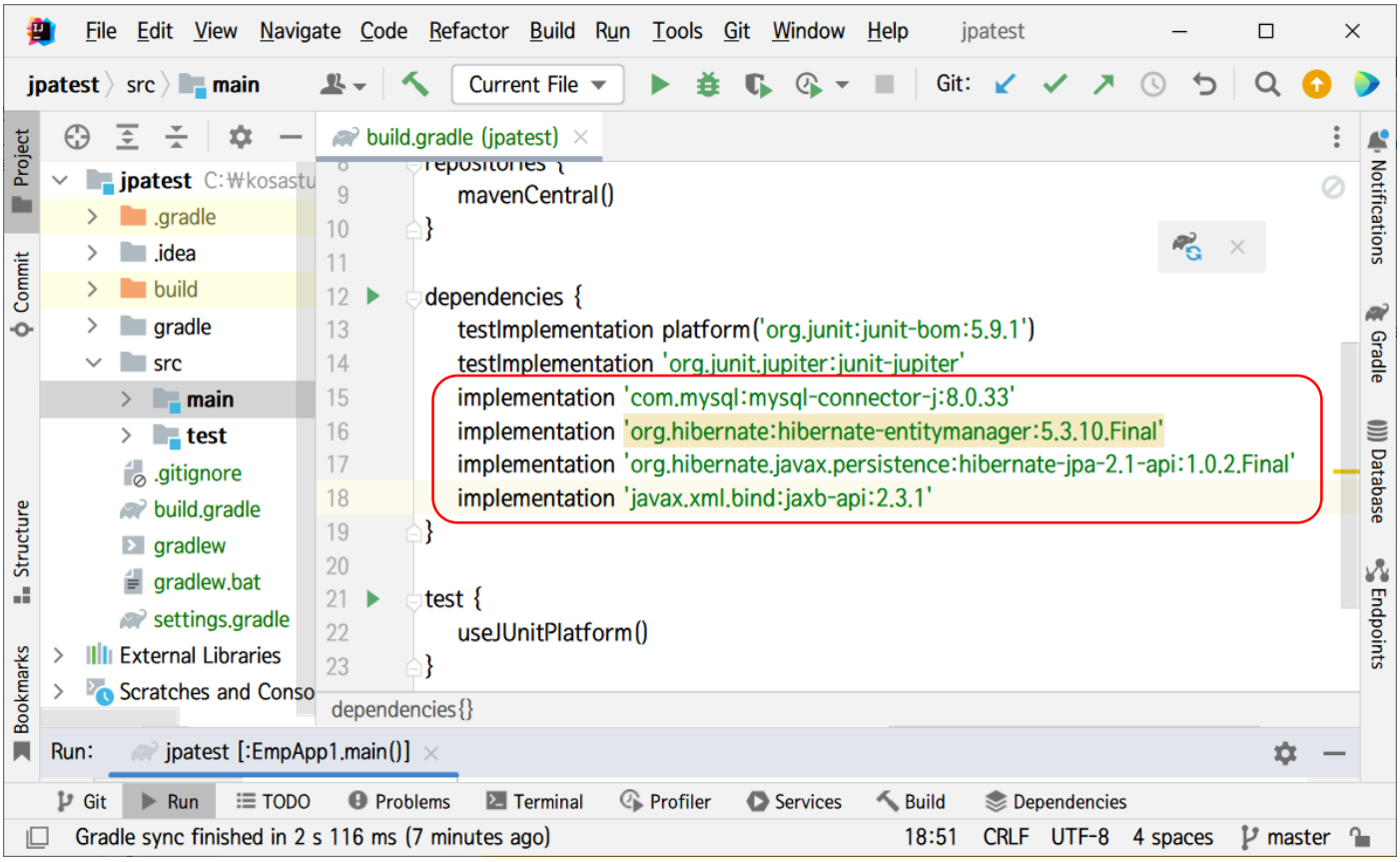


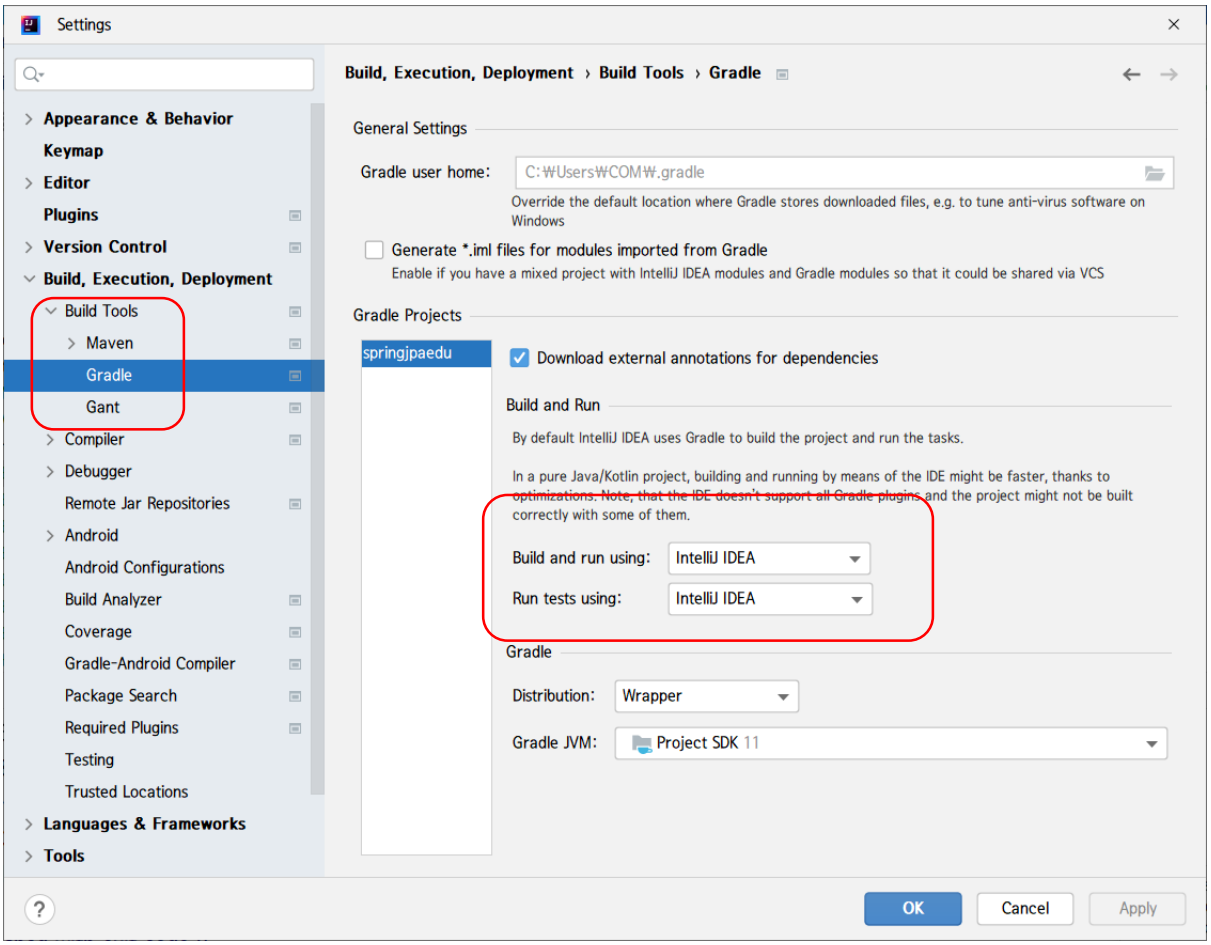

Maven Repository: com.mysql
mvnrepository.com/artifact/com.mysql/mysql-connector-j/8.0.33
MVN REPOSITORY
Search for groups, artifacts, categories
Search
Categories | Popular | Contact Us
Indexed Artifacts (33.6M)
Popular Categories
Testing Frameworks & Tools
Android Packages
Logging Frameworks
Java Specifications
JSON Libraries
JVM Languages
Core Utilities
Mocking
Language Runtime
Web Assets
Annotation Libraries
Logging Bridges
HTTP Clients
Dependency Injection
Home » com.mysql » mysql-connector-j » 8.0.33
MySQL Connector/J » 8.0.33
JDBC Type 4 driver for MySQL.
Categories JDBC Drivers
Tags database sql jdbc driver connector mysql
Organization Oracle Corporation
HomePage http://dev.mysql.com/doc/connector-j/en/
Date Apr 18, 2023
Files pom (3 KB) jar (2.4 MB) View All
Repositories Central
Ranking #2207 in MvnRepository (See Top Artifacts)
#13 in JDBC Drivers
Used By 190 artifacts
Maven Gradle Gradle (Short) Gradle (Kotlin) SBT Ivy Grape Leiningen Buildr
// https://mvnrepository.com/artifact/com.mysql/mysql-connector-j
implementation 'com.mysql:mysql-connector-j:8.0.33'
Include comment with link to declaration

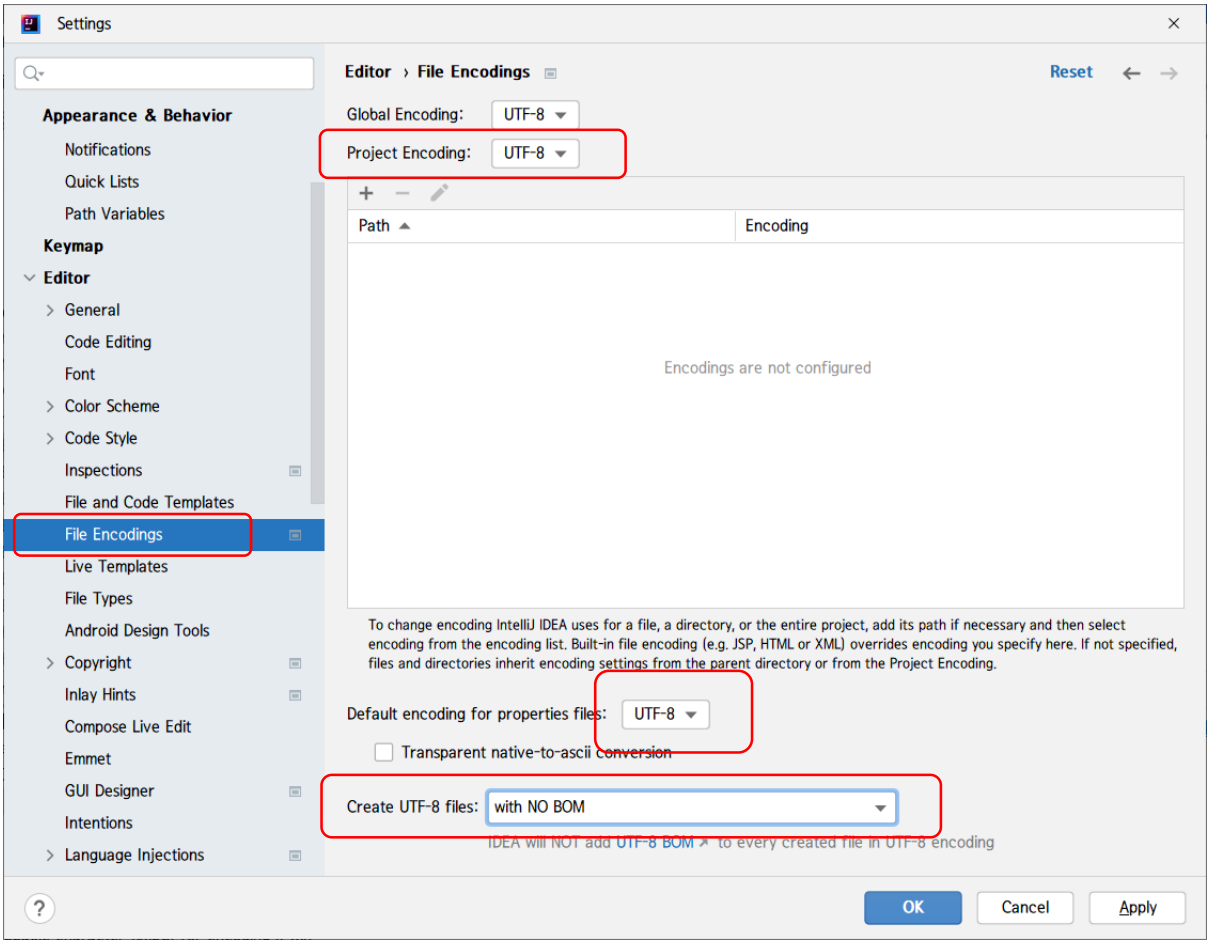

Maven Repository: org.hibernat
mvnrepository.com/artifact/org.hibernate/hibernate-entitymanager/5.3.10.Final
Popular Categories
Testing Frameworks & Tools
Android Packages
Logging Frameworks
Java Specifications
JSON Libraries
JVM Languages
Core Utilities
Mocking
Language Runtime
Web Assets
Annotation Libraries
Logging Bridges
HTTP Clients
Dependency Injection
XML Processing
Web Frameworks
I/O Utilities
Defect Detection Metadata
Hibernate EntityManager Relocation » 5.3.10.Final
Hibernate ORM 6.0.0.Alpha7 release. See http://hibernate.org/orm/releases/6.0
License LGPL 2.1
Categories JPA Implementations
Tags hibernate persistence jpa
Organization Hibernate.org
HomePage http://hibernate.org/orm
Date Apr 20, 2019
Files pom (4 KB) jar (591 bytes) View All
Repositories Central
Ranking #156 in MvnRepository (See Top Artifacts)
#1 in JPA Implementations
Used By 2,839 artifacts
Vulnerabilities Vulnerabilities from dependencies:
CVE-2020-25638
CVE-2020-10683
Note: There is a new version for this artifact
New Version 6.0.0.Alpha7
Maven Gradle Gradle (Short) Gradle (Kotlin) SBT Ivy Grape Leiningen Buildr
// https://mvnrepository.com/artifact/org.hibernate/hibernate-entitymanager
implementation 'org.hibernate:hibernate-entitymanager:5.3.10.Final'


Maven Repository: org.hibernate
mvnrepository.com/artifact/org.hibernate.javax.persistence/hibernate-jpa-2.1-api/1.0....
MVN REPOSITORY
Search for groups, artifacts, categories
Search
Categories | Popular | Contact Us
Indexed Artifacts (33.6M)
Popular Categories
Testing Frameworks & Tools
Android Packages
Logging Frameworks
Java Specifications
JSON Libraries
JVM Languages
Core Utilities
Mocking
Language Runtime
Web Assets
Annotation Libraries
Logging Bridges
HTTP Clients
Dependency Injection
Home » org.hibernate.javax.persistence » hibernate-jpa-2.1-api » 1.0.2.Final
Java Persistence API, Version 2.1 » 1.0.2.Final
Clean-room definition of JPA APIs intended for use in developing Hibernate JPA implementation. See README.md for details
License EDL 1.0 EPL 1.0
Tags persistence jpa hibernate api
HomePage http://hibernate.org
Date Jan 23, 2018
Files pom (2 KB) jar (110 KB) View All
Repositories Central JBoss Releases Mulesoft
Ranking #433 in MvnRepository (See Top Artifacts)
Used By 1,024 artifacts
Maven Gradle Gradle (Short) Gradle (Kotlin) SBT Ivy Grape Leiningen Buildr
// https://mvnrepository.com/artifact/org.hibernate.javax.persistence/hibernate-jpa-2.1-api
implementation 'org.hibernate.javax.persistence:hibernate-jpa-2.1-api:1.0.2.Final'
Include comment with link to declaration
Ads by Google


Maven Repository: javax.xml.bin
mvnrepository.com/artifact/javax.xml.bind/jaxb-api/2.3.1
MVN REPOSITORY
Search for groups, artifacts, categories
Search
Categories | Popular | Contact Us
Indexed Artifacts (33.6M)
Popular Categories
Testing Frameworks & Tools
Android Packages
Logging Frameworks
Java Specifications
JSON Libraries
JVM Languages
Core Utilities
Mocking
Language Runtime
Web Assets
Annotation Libraries
Logging Bridges
HTTP Clients
Dependency Injection
Home » javax.xml.bind » jaxb-api » 2.3.1
JAXB API » 2.3.1
JAXB provides an API and tools that automate the mapping between XML documents and Java objects.
License CDDL 1.1
Categories XML Processing Java Specifications
Tags binding standard javax jaxb xml api specs
Date Oct 02, 2018
Files pom (19 KB) jar (125 KB) View All
Repositories Central JCenter Redhat GA
Ranking #82 in MvnRepository (See Top Artifacts)
#1 in XML Processing
#6 in Java Specifications
Used By 5,633 artifacts
Note: There is a new version for this artifact
New Version 2.4.0-b180830.0359
Maven Gradle Gradle (Short) Gradle (Kotlin) SBT Ivy Grape Leiningen Buildr
// https://mvnrepository.com/artifact/javax.xml.bind/jaxb-api
implementation 'javax.xml.bind:jaxb-api:2.3.1'


File Edit View Navigate Code Refactor Build Run Tools Git Window Help jpatest
jpatest src main
Current File
Git:
build.gradle (jpatest)
jpatest C:₩kosastu
.gradle
.idea
build
gradle
src
main
test
.gitignore
build.gradle
gradlew
gradlew.bat
settings.gradle
External Libraries
Scratches and Conso
mavenCentral()
}
dependencies {
testImplementation platform('org.junit:junit-bom:5.9.1')
testImplementation 'org.junit.jupiter:junit-jupiter'
implementation 'com.mysql:mysql-connector-j:8.0.33'
implementation 'org.hibernate:hibernate-entitymanager:5.3.10.Final'
implementation 'org.hibernate.javax.persistence:hibernate-jpa-2.1-api:1.0.2.Final'
implementation 'javax.xml.bind:jaxb-api:2.3.1'
}
test {
useJUnitPlatform()
}
dependencies{}
Run: jpatest [:EmpApp1.main()]
Git Run TODO Problems Terminal Profiler Services Build Dependencies
Gradle sync finished in 2 s 116 ms (7 minutes ago)
18:51 CRLF UTF-8 4 spaces master
Project
Commit
Structure
Bookmarks
Notifications
Gradle
Database
Endpoints

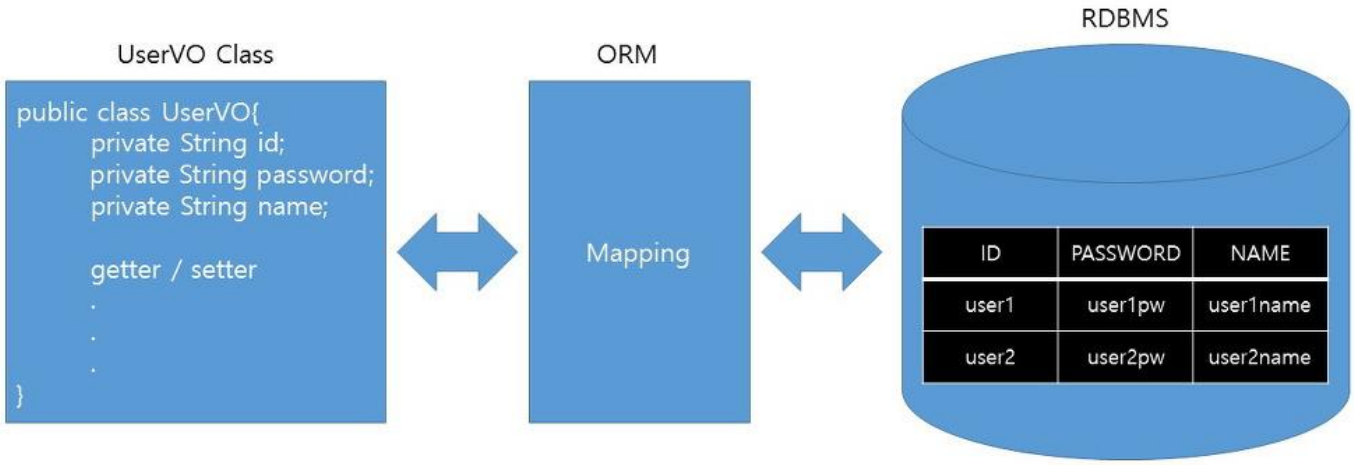

Settings
Build, Execution, Deployment › Build Tools › Gradle
Appearance & Behavior
Keymap
Editor
Plugins
Version Control
Build, Execution, Deployment
Build Tools
Maven
Gradle
Gant
Compiler
Debugger
Remote Jar Repositories
Android
Android Configurations
Build Analyzer
Coverage
Gradle-Android Compiler
Package Search
Required Plugins
Testing
Trusted Locations
Languages & Frameworks
Tools
General Settings
Gradle user home:
C:₩Users₩COM₩.gradle
Override the default location where Gradle stores downloaded files, e.g. to tune anti-virus software on Windows
Generate *.iml files for modules imported from Gradle
Enable if you have a mixed project with IntelliJ IDEA modules and Gradle modules so that it could be shared via VCS
Gradle Projects
springjpaedu
Download external annotations for dependencies
Build and Run
By default IntelliJ IDEA uses Gradle to build the project and run the tasks.
In a pure Java/Kotlin project, building and running by means of the IDE might be faster, thanks to optimizations. Note, that the IDE doesn't support all Gradle plugins and the project might not be built correctly with some of them.
Build and run using: IntelliJ IDEA
Run tests using: IntelliJ IDEA
Gradle
Distribution: Wrapper
Gradle JVM: Project SDK 11
OK
Cancel
Apply



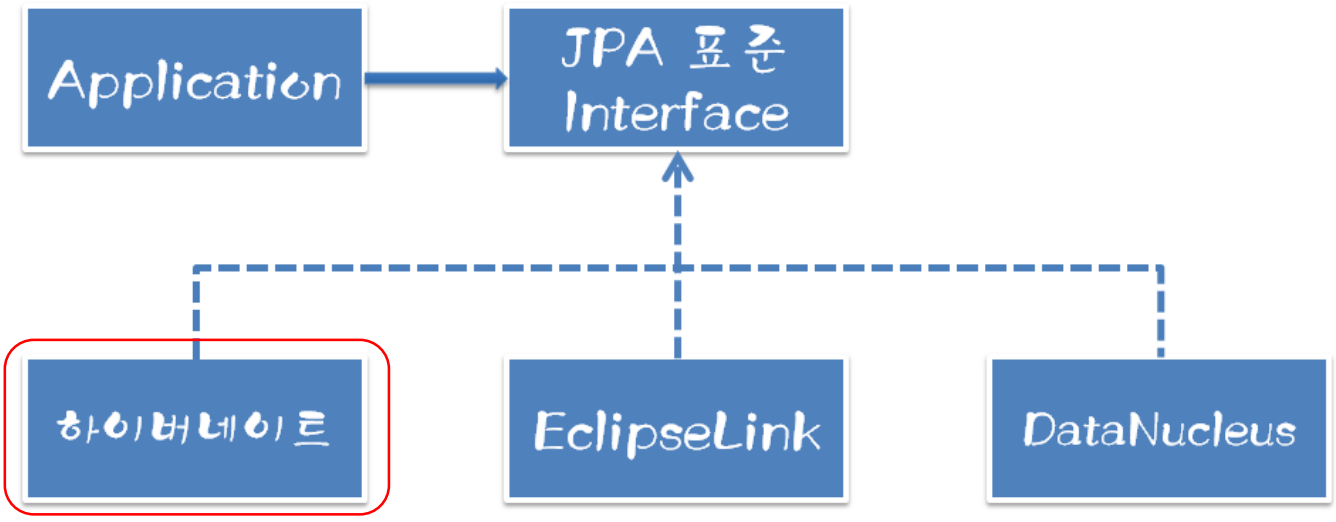
1. JPA 개념

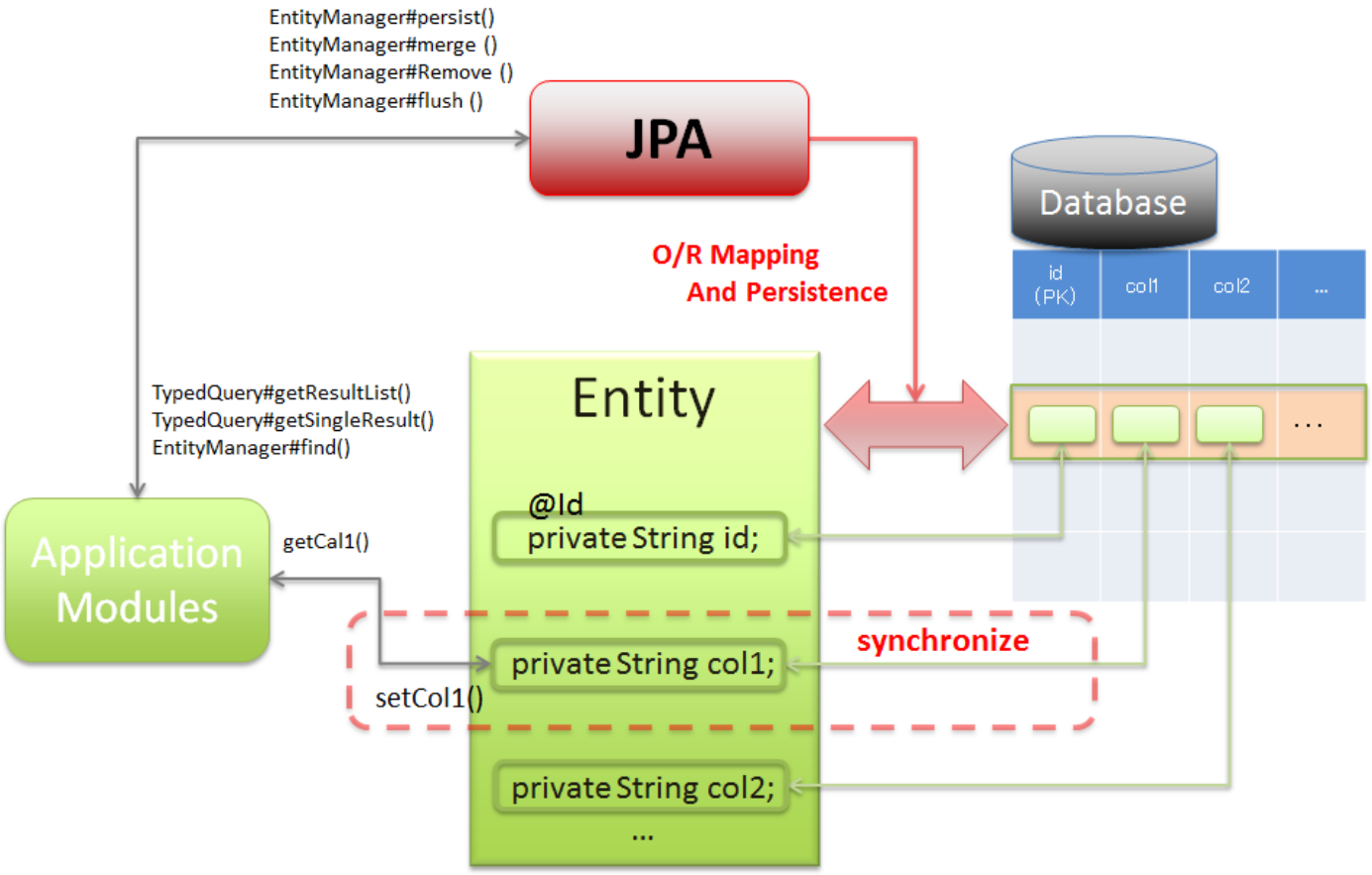
JPA는 Java Persistence API 의 약자로서, RDBMS와 OOP 객체 사이의 불일치에서 오는 패러다임을 해결하기 위해서 만들어진 ORM(Object-Relational Mapping) 기술이다.

ORM이란 Object Relational Mapping, 객체-관계 매핑의 줄임말로서 OOP의 객체 구현 클래스와 RDBMS에서 사용하는 테이블을 자동으로 매핑하는 것을 의미한다. 이 때, 클래스와 테이블은 기존부터 호환 가능성을 두고 만들어진 것이 아니므로 불일치가 발생하는데 이를 ORM을 통해서 객체 간의 관계를 바탕으로 SQL문을 자동 생성하여 불일치를 해결한다. 이 방법을 통해서 SQL문을 구현할 필요없이 객체를 통해 간접적으로 데이터베이스를 조작할 수 있다.



JPA 는 자바 ORM에 대한 API 표준 명세이고, 인터페이스의 모음이다. 따라서 구현체가 없으므로, 사용하기 위해서는 ORM프레임워크를 선택해야 한다. 다양한 프레임워크가 존재하지만 가장 대중적인 것은 하이버네이트이다.





- Persistence Framework

JDBC 프로그래밍에서 경험하게 되는 복잡함이나 번거로움 없이 간단한 작업만으로 데이터베이스와 연동되는 시스템을 빠르게 개발할 수 있다. 일반적으로 SQL Mapper와 ORM으로 나눠진다.

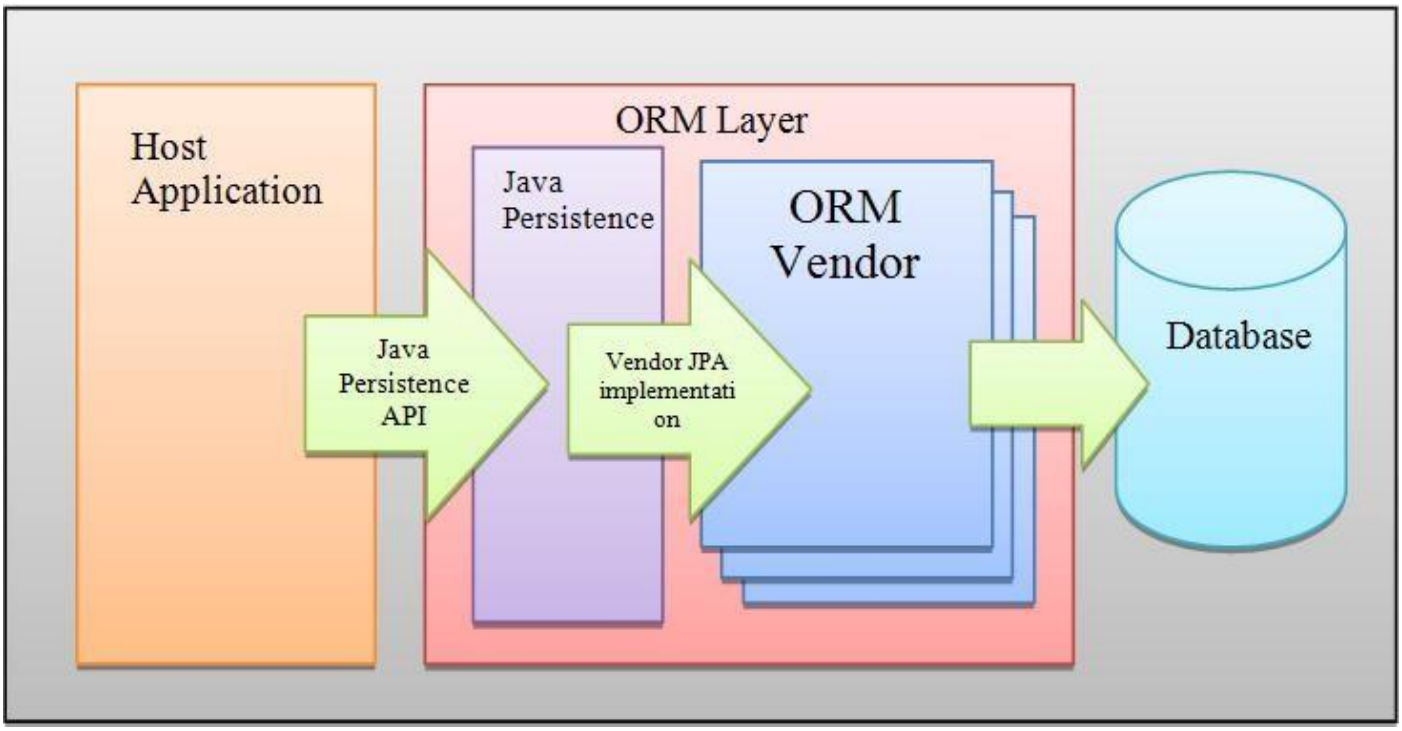
[SQL Mapper]

SQL <-> SQL Mapper <-> Object 필드
직접 작성한 SQL 문장으로 데이터베이스 데이터를 다룬다.
Mybatis, JdbcTemplates(Spring)

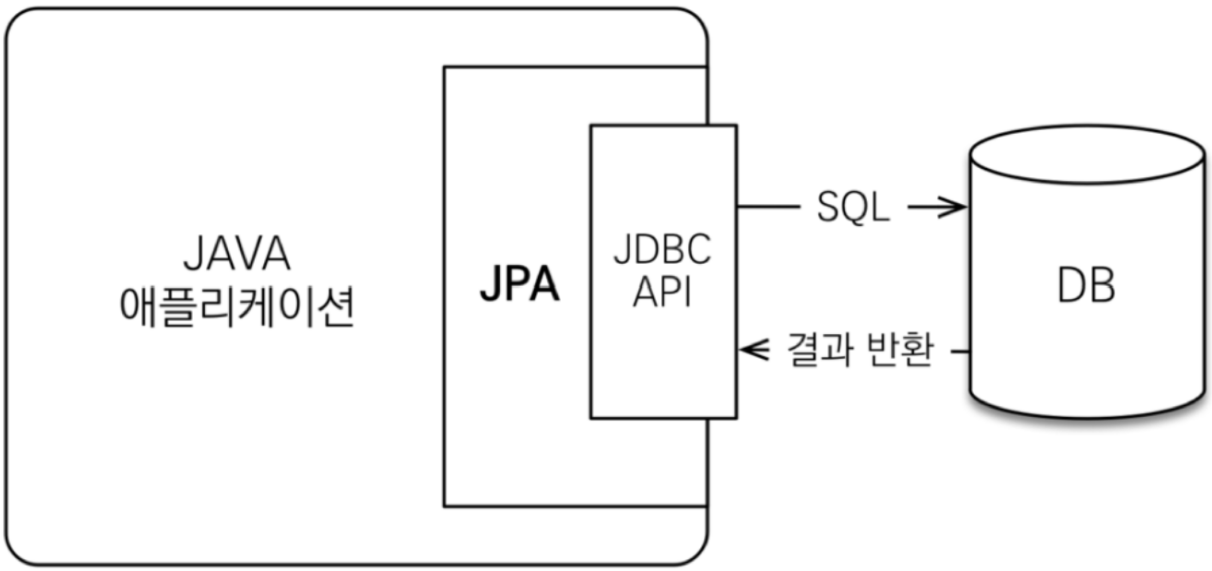
[ORM]

Database data <-> ORM <-> Object 필드
객체를 통해서 간접적으로 데이터베이스의 데이터를 다룬다.
객체와 관계형 데이터베이스의 데이터를 자동으로 맵핑시킨다.
JPA, Hibernate 등

- 동작 과정



JPA는 애플리케이션과 JDBC 사이에서 동작한다. JPA 내부에서 JDBC API를 사용하여 SQL을 호출하여 DB와 통신한다. 개발자가 ORM 프레임워크에 저장하면 적절한 INSERT SQL을 생성해 데이터베이스에 저장해주고, 검색을 하면 적절한 SELECT SQL을 생성해 결과를 객체에 매핑하고 전달한다.



- JPA 의 사용 이점

1. 생산성

JPA를 사용하면 자바 컬렉션에 저장하듯이 JPA에게 저장할 객체를 전달하면 된다.

지루하고 반복적인 코드를 개발자가 직접 작성하지 않아도 되며, DDL문도 자동으로 생성해주기 때문에 데이터베이스 설계 중심을 객체 설계 중심으로 변경할 수 있다.

2. 유지보수

필드를 하나만 추가해도 관련된 SQL과 JDBC 코드를 전부 수행해야 했지만 JPA는 이를 대신 처리해주기 때문에 개발자가 유지보수해야 하는 코드가 줄어든다.

3. 패러다임의 불일치 해결

JPA는 연관된 객체를 사용하는 시점에 SQL을 전달할 수 있고, 같은 트랜잭션 내에서 조회할 때 동일성도 보장하기 때문에 다양한 패러다임의 불일치를 해결한다.

4. 성능

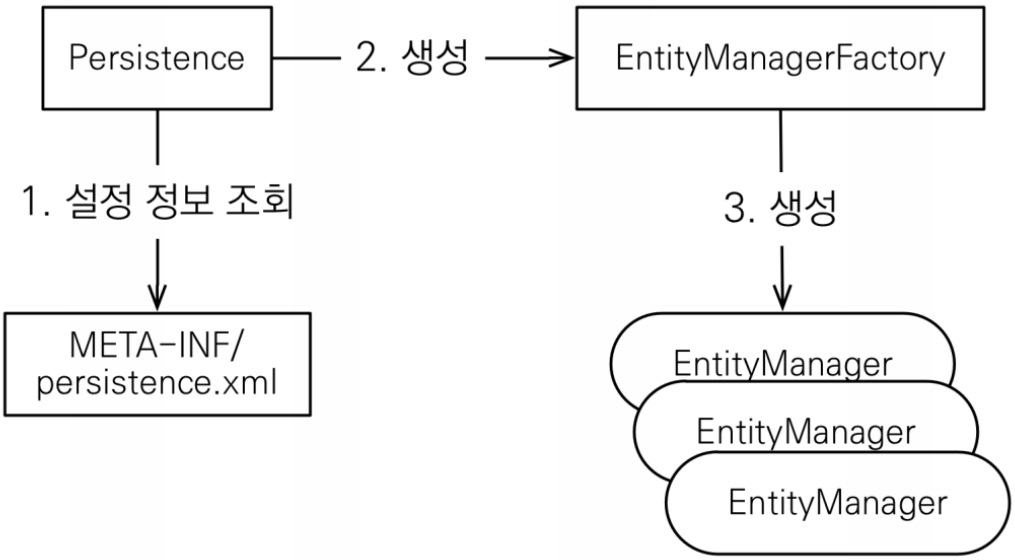
애플리케이션과 데이터베이스 사이에서 성능 최적화 기회를 제공한다.

같은 트랜잭션안에서는 같은 엔티티를 반환하기 때문에 데이터 베이스와의 통신 횟수를 줄일 수 있다. 또한, 트랜잭션을 commit하기 전까지 메모리에 쌓고 한 번에 SQL을 전송한다.

5. 데이터 접근 추상화와 벤더 독립성

RDB는 같은 기능이라도 벤더마다 사용법이 다르기 때문에 처음 선택한 데이터베이스에 종속되고 변경이 어렵다. JPA는 애플리케이션과 데이터베이스 사이에서 추상화된 데이터 접근을 제공하기 때문에 종속이 되지 않도록 한다. 만약 DB가 변경되더라도 JPA에게 알려주면 간단하게 변경이 가능하다.

2. JPA 프로그래밍



1. persistence.xml 파일을 통해서 JPA를 설정한다
2. EntityManagerFactory를 생성한다
3. EntityManager를 생성하여 Entity를 영속성 컨텍스트(Persistence Context)를 통해 관리한다.

```
EntityManagerFactory factory = Persistence.createEntityManagerFactory("emptest");
EntityManager em = factory.createEntityManager();
```

EntityManagerFactory

데이터베이스와 상호 작용을 위한 EntityManager 객체를 생성하기 위해 사용되는 객체로서 애플리케이션에서 한 번만 생성하고 공유해서 사용한다.

Thread-Safe 하므로 여러 스레드에서 동시에 접근해도 안전하다.

EntityManagerFactory 객체를 통해 생성되는 모든 EntityManager 객체는 동일한 데이터베이스에 접속한다.

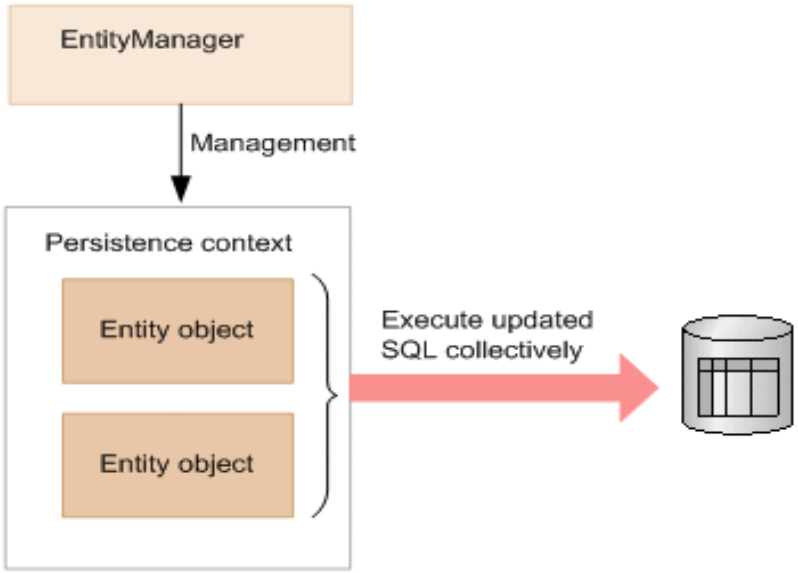
EntityManager

Entity를 관리하는 객체이다

데이터베이스에 대한 CRUD 작업은 모두 영속성 컨텍스트를 사용하는 EntityManager 객체를 통해 이루어진다.

동시성의 문제가 발생할 수 있으니 스레드 간에 공유하지 않는다,

모든 데이터 변경은 트랜잭션 안에서 이루어져야 한다. (트랜잭션을 시작하고 처리해야 함) -------- 데이터베이스의 논리적인 작업 단위



다음과 같은 메서드들이 제공된다.

flush()

영속성 컨텍스트(Persistence Context)의 변경 내용을 데이터베이스에 반영한다.

일반적으로는 flush() 메서드를 직접 사용하지는 않고, 자바 애플리케이션에서 커밋 명령이 들어왔을 때 자동으로 실행된다.

detach()

특정 Entity를 준영속 상태(영속 컨텍스트의 관리를 받지 않음)로 바꾼다.

clear()

Persistence Context를 초기화한다.

close()

Persistence Context를 종료한다.

merge()

준영속 상태의 엔티티를 이용해서 새로운 영속 상태의 엔티티를 반환한다.

find()

식별자 값을 통해 Entity를 찾는다. (DB 테이블의 데이터 또는 행을 찾는다.)

persist()

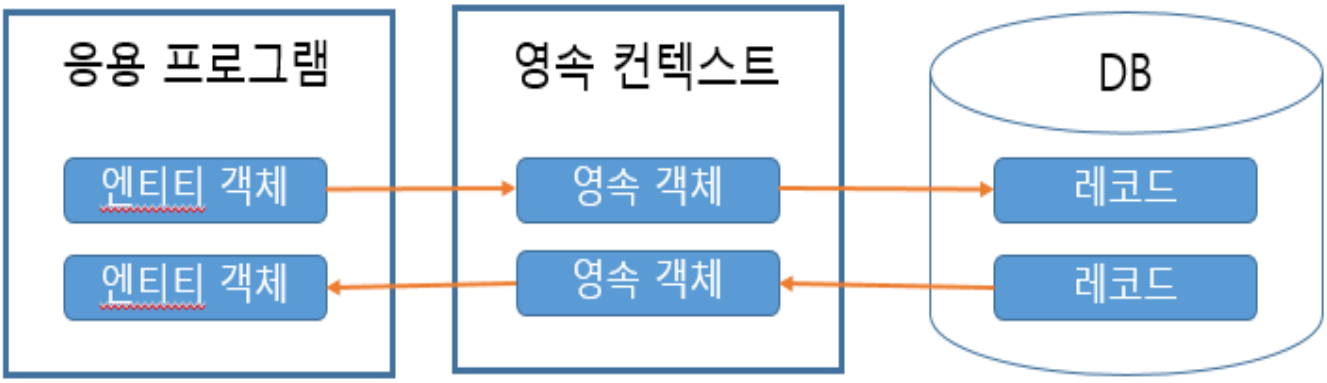
생성된 Entity 객체를 영속성 컨텍스트(Persistence Context)에 저장한다.

remove()

식별자 값을 통해 영속성 컨텍스트(Persistence Context)에서 Entity 객체를 삭제한다.

엔티티(Entity)

JPA에서 엔티티란 DB 테이블에 대응하는 하나의 객체라고 생각할 수 있다.
@Entity 어노테이션이 붙은 클래스를 JPA에서는 엔티티라고 부르며, 이 엔티티는 영속성 컨텍스트에 담겨 EntityManager에 의해서 관리된다.



```
@Entity
public class EntityTest {
	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	private int id;
	private String name;
	private int age;
	private LocalDateTime   birthday;

	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
```

영속성 컨텍스트(persistence context)

- 어플리케이션과 데이터베이스 사이에 존재하는 논리적인 개념으로 엔티티를 저장하는 환경이다.

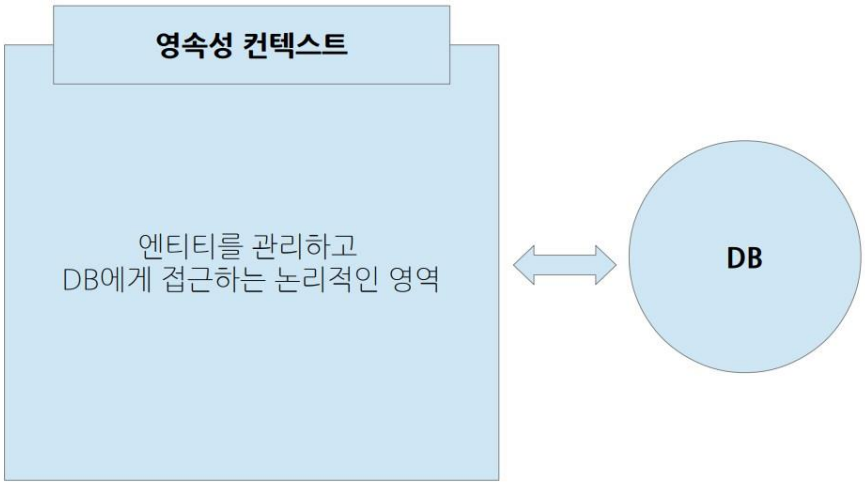
- EntityManager 객체를 통해서만 접근이 가능하다.

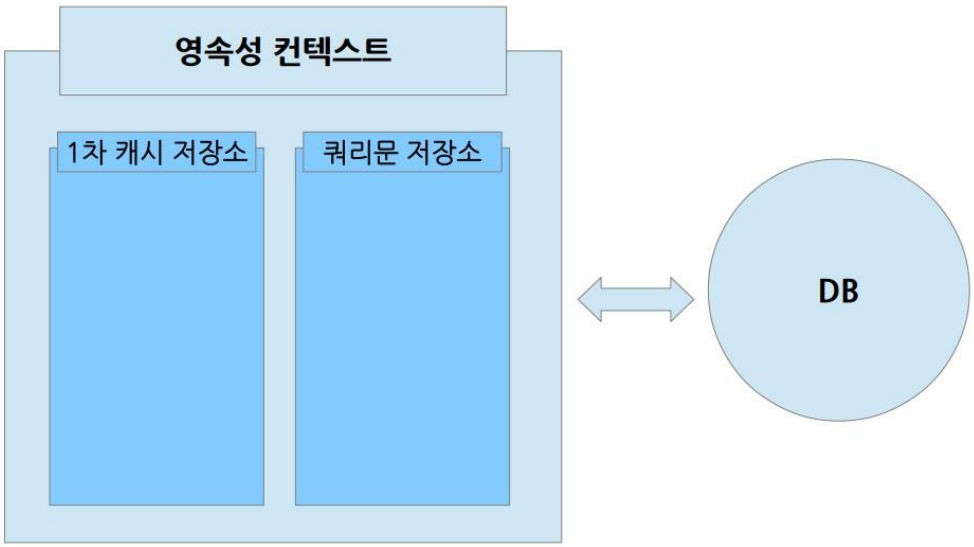
- 영속성 컨텍스트에 존재하는 엔티티는 플러시 호출 시 데이터베이스에 반영된다.

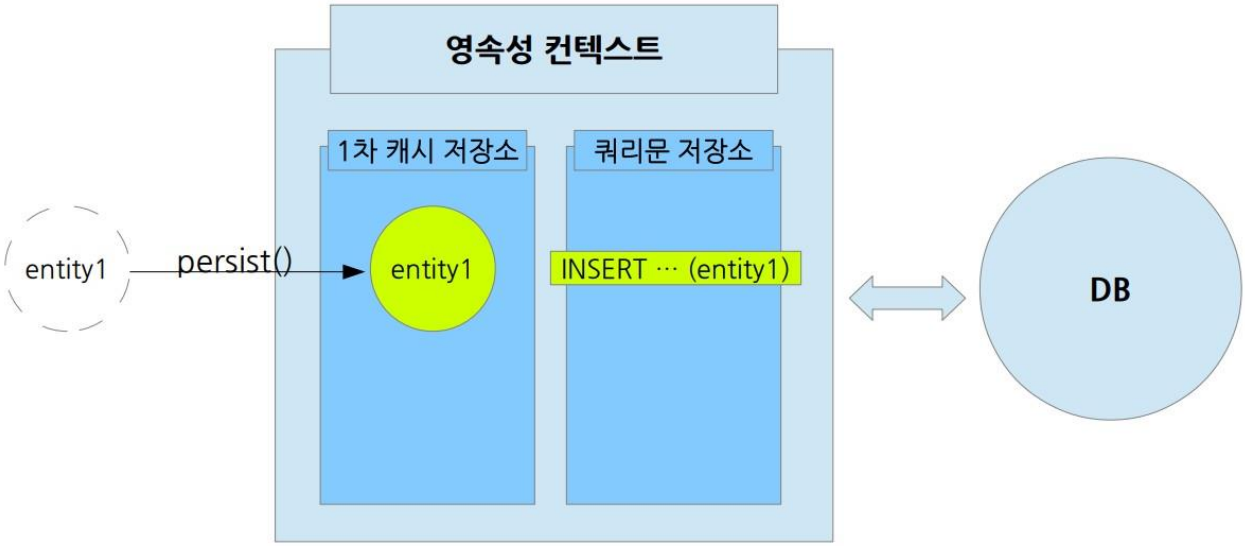
entityManger.flush() 로 플러시 직접 호출

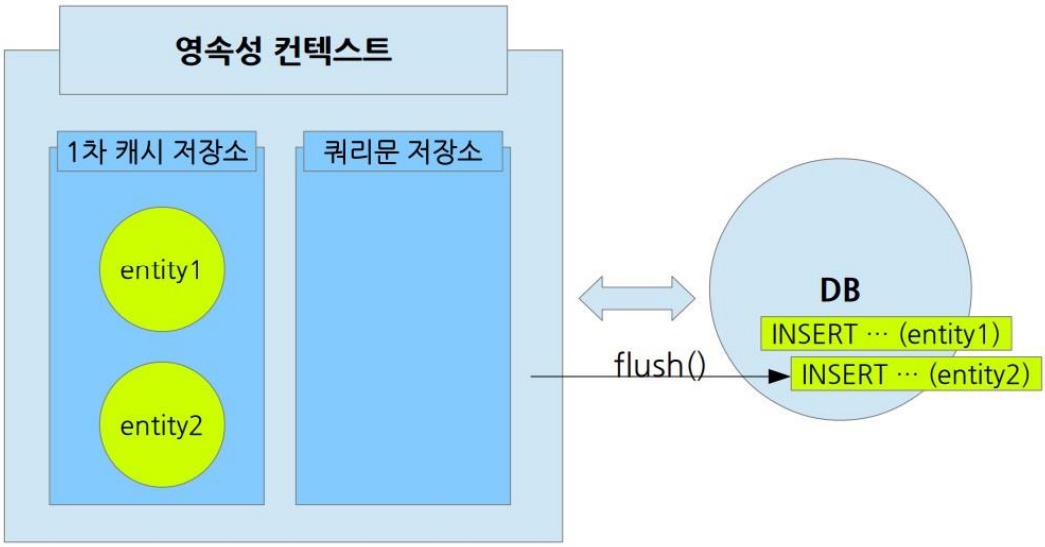
트랜젝션 커밋(commit) 시 플러시 자동 호출

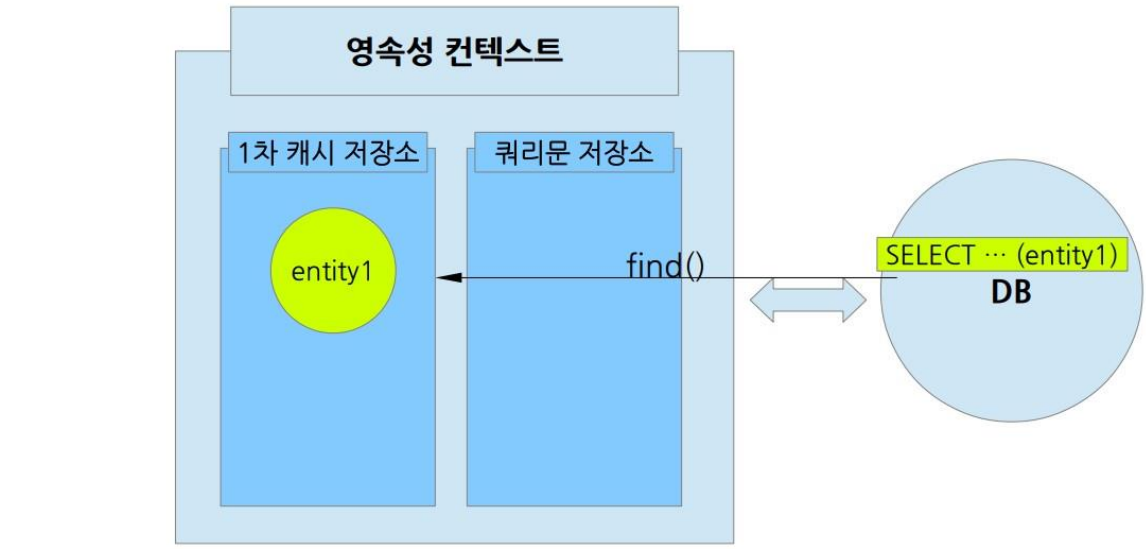
JPQL 쿼리 실행 시 플러시 자동 호출

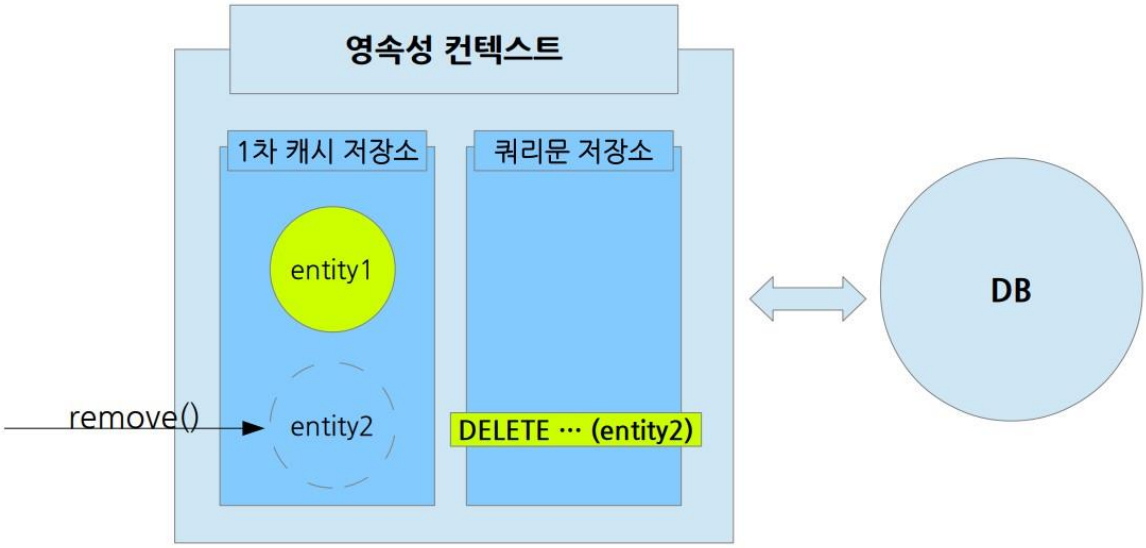


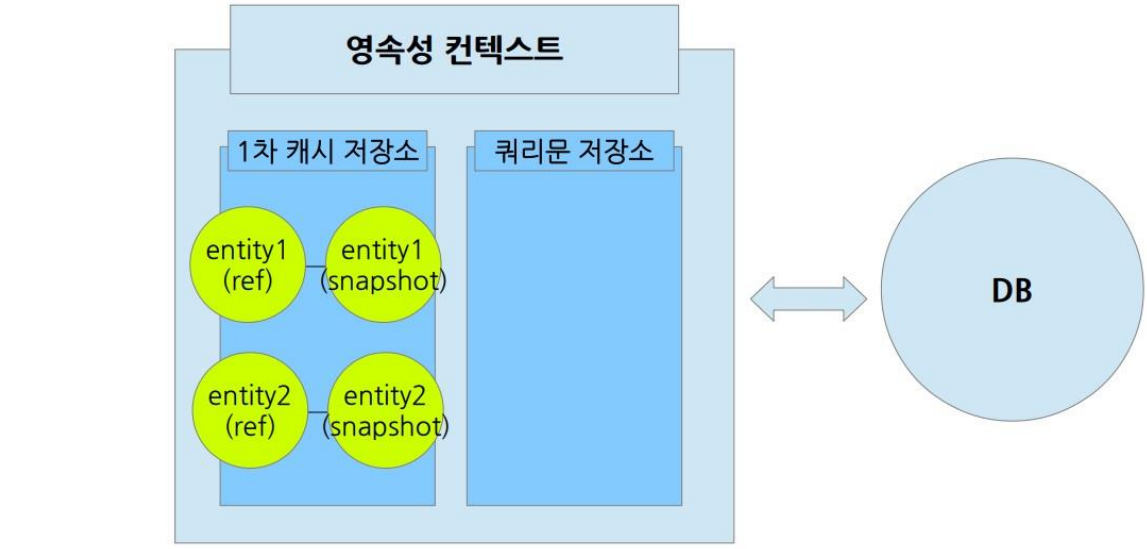



영속성 컨텍스트
1차 캐시 저장소
쿼리문 저장소
entity1
persist()
entity1
INSERT … (entity1)
DB


영속성 컨텍스트
1차 캐시 저장소
쿼리문 저장소
entity1
entity2
DB
INSERT … (entity1)
flush()
INSERT … (entity2)


영속성 컨텍스트
1차 캐시 저장소
쿼리문 저장소
entity1
find()
SELECT … (entity1)
DB

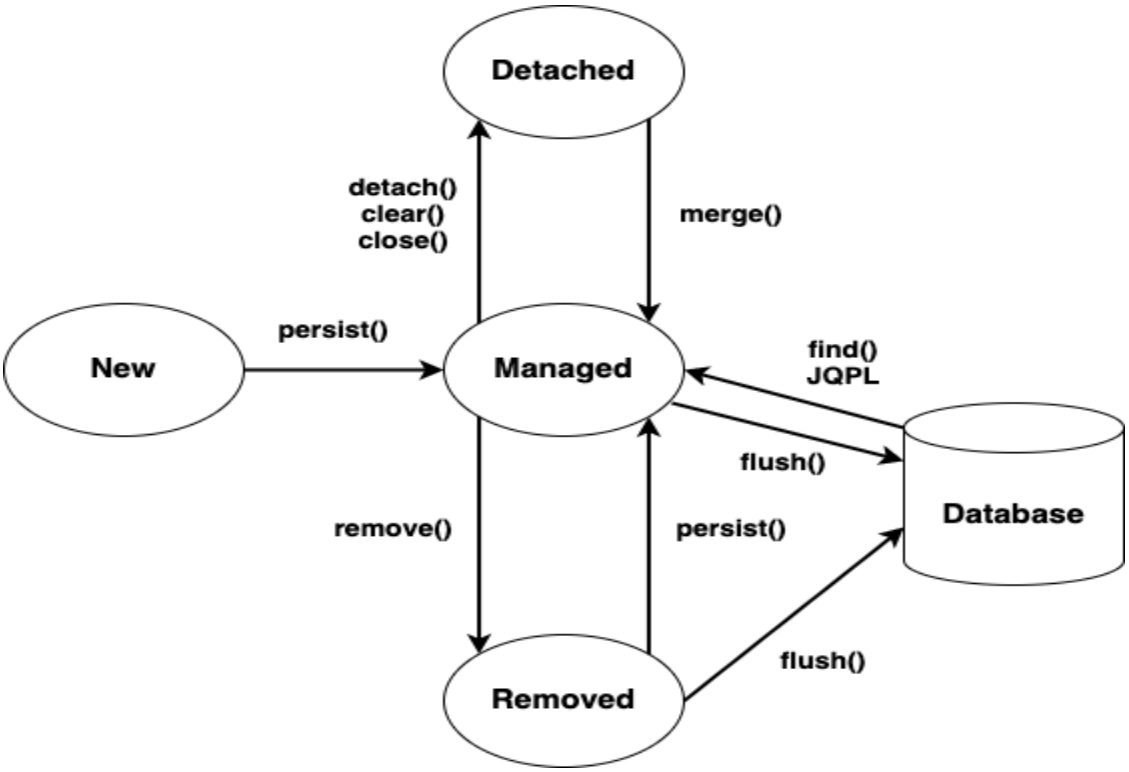

영속성 컨텍스트
1차 캐시 저장소
쿼리문 저장소
entity1
remove()
entity2
DELETE … (entity2)
DB


영속성 컨텍스트
1차 캐시 저장소
쿼리문 저장소
entity1 (ref)
entity1 (snapshot)
entity2 (ref)
entity2 (snapshot)
DB

- 엔티티 생명주기

Entity에는 아래와 같은 4가지의 상태가 존재한다.

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



- 엔티티 생성과 저장

1. 자바 어플리케이션에서 어떤 엔티티 객체를 생성하여 JPA에게 데이터베이스 저장을 부탁하면,
2. 만들어진 엔티티는 1차적으로 영속성 컨텍스트에 저장된다. 1차 캐시 정도라고 생각하면 된다.

그리고, 저장한 엔티티를 데이터베이스에 저장하기 위한 쿼리문을 생성시켜 쓰기 지연 SQL 저장

소에 저장한다. 계속해서 엔티티를 넘기면 엔티티들과 쿼리문들은 차곡차곡

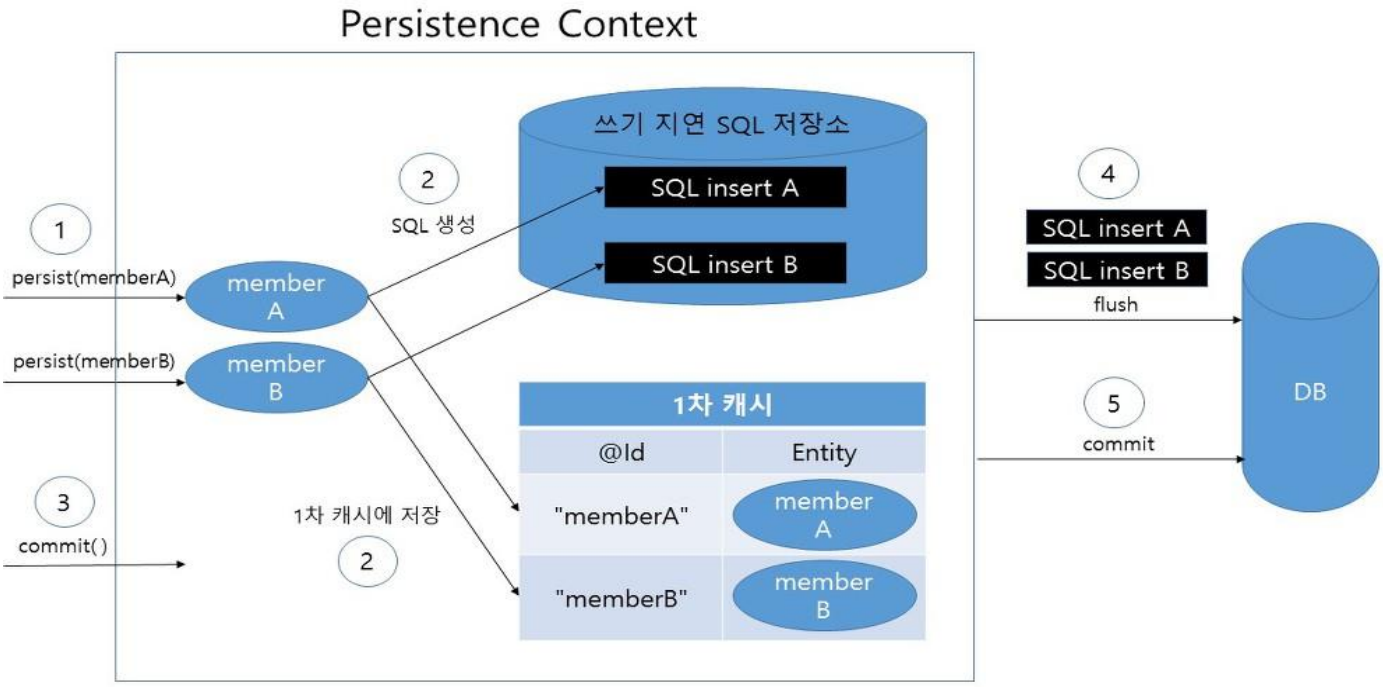
영속성 컨텍스트에
저장된다.

3. 자바 어플리케이션에서 커밋 명령이 내려지면 영속 컨텍스트에는 자동으로 flush( )가 호출되고,

4. 영속성 컨텍스트의 변경내용을 데이터베이스와 동기(flush)화 한다(SQL 저장소의 쿼리를 실행시
킨다).

5. 마지막으로 데이터베이스에게 commit 쿼리문을 명령한다.

```
EntityManager em = factory.createEntityManager();
try {
        em.getTransaction().begin();
        em.persist(vo);
        em.getTransaction().commit();
} catch (Exception e) {
        result = false;
```



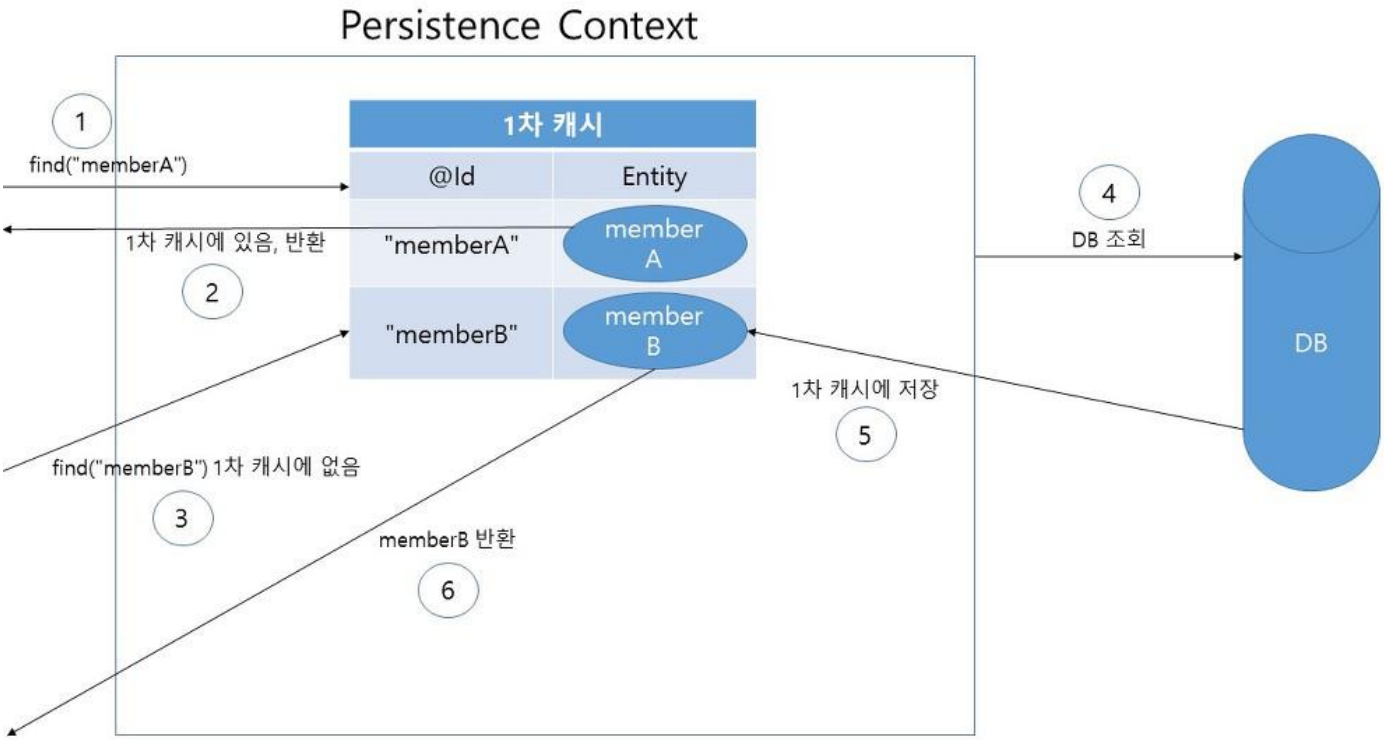
- 엔티티 조회

1. 자바 어플리케이션에서 JPA에게 데이터베이스 조회를 부탁하면, 1차적으로 영속성 컨텍스트에서
   엔티티를 찾는다.
2. 있으면 자바 어플리케이션에 엔티티를 넘긴다.
3. 영속성 컨텍스트에 없는 엔티티 조회를 부탁하면
4. 쿼리문을 사용해 데이터베이스에서 찾아와
5. 영속성 컨텍스트에 엔티티로 저장하고
6. 자바 어플리케이션에 그 엔티티를 넘긴다.

```
EntityManager em = factory.createEntityManager();
TypedQuery<Visitor> q = em.createQuery("select t from Visitor t", Visitor.class);
List<Visitor> list = q.getResultList();
```

```
EntityManager em = factory.createEntityManager();
TypedQuery<Visitor> q = em.createQuery(
                        "select t from Visitor t WHERE t.memo like :keyword", Visitor.class);
q.setParameter("keyword", "%" + keyword + "%");
List<Visitor> list = q.getResultList();
```

```
EntityManager em = factory.createEntityManager();
Visitor vo = em.find(Visitor.class, id);
```

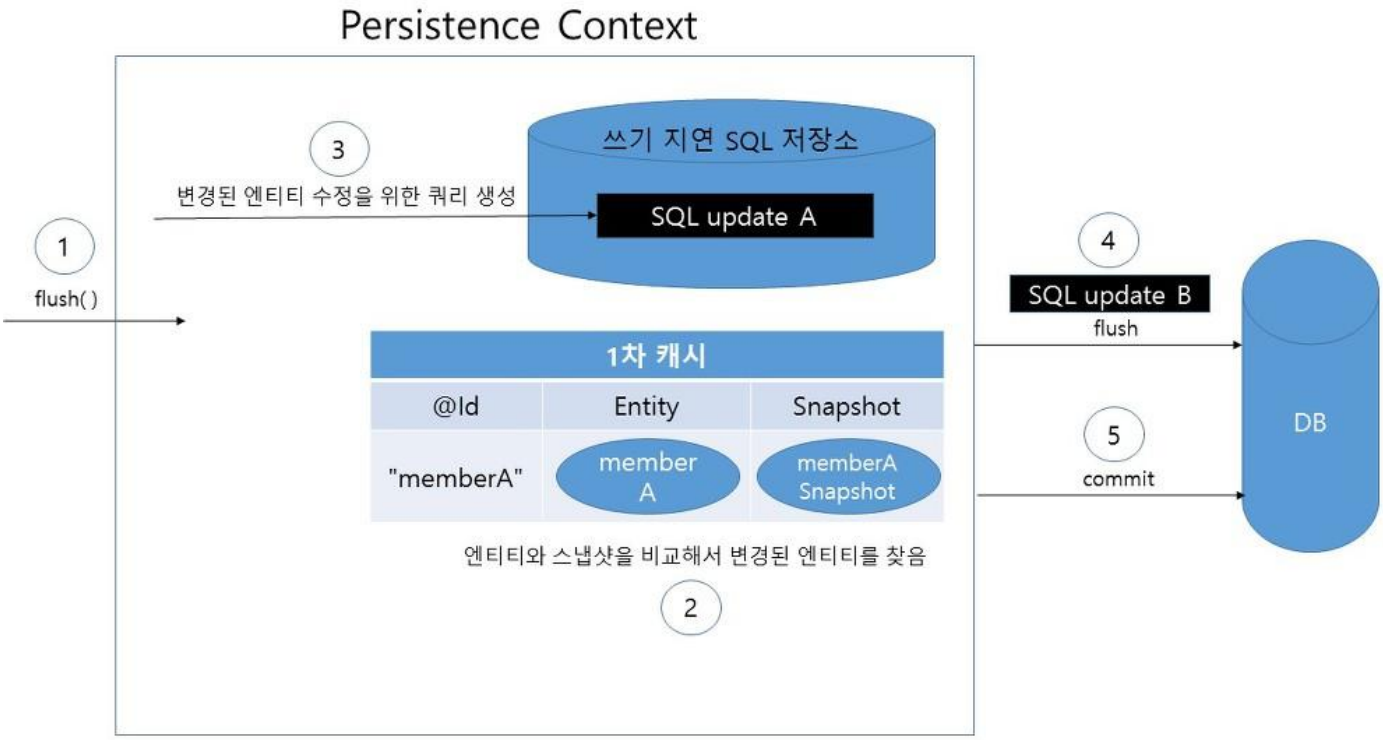


- 엔티티 변경

JPA는 엔티티를 영속성 컨텍스트에 보관할 때, 최초의 상태를 복사해서 저장해 두는데,
이것을 스냅샷이라 한다.

1. 자바 어플리케이션에서 커밋 명령이 들어오면, 영속 컨텍스트에는 자동으로 flush( )가 호출되고,
2. 엔티티와 스냅샷을 비교해서 변경된 엔티티를 찾는다.
3. 변경된 엔티티가 있으면 데이터베이스에 변경사항을 저장하기 위해 쿼리를 생성하고,
4. 영속성 컨텍스트의 변경내용을 데이터베이스와 동기(flush)화 한다(SQL 저장소의 쿼리를 실행시
   킨다).
5. 마지막으로 데이터베이스에게 commit 쿼리문을 명령한다.

```
EntityManager em = factory.createEntityManager();
try {
        em.getTransaction().begin();
        Visitor oldVo = em.find(Visitor.class, vo.getId());
        System.out.println(oldVo.getName());
        oldVo.setName(vo.getName());
        oldVo.setMemo(vo.getMemo());
        em.getTransaction().commit();
} catch (Exception e) {
```



이렇게 엔티티의 변경사항을 데이터베이스에 자동으로 반영하는 기능을 변경감지(Dirty Checking) 라 한다.

- 엔티티 삭제

앞의 과정과 마찬가지로, 자바 어플리케이션에서 엔티티 삭제 명령이 들어오면, 엔티티를 찾고 쓰기 지연 SQL 저장소에 delete 쿼리를 생성한다.
그리고 자바 어플리케이션에서 커밋 명령이 들어오면, 자동으로 flush( )가 호출되고,
영속성 컨텍스트의 변경내용을 데이터베이스와 동기(flush)화 한다(SQL 저장소의 쿼리를 실행시킨다).
마지막으로 데이터베이스에게 commit 쿼리문을 명령한다.

[영속성 컨텍스트의 장점]

- 캐시 사용

엔티티 조회시 영속성 컨테스트에 존재하면 바로 리턴, 없으면 데이터베이스 조회 후 리턴.

- 동일성(==)보장

조회 시 항상 같은 엔티티 인스턴스를 리턴(주소값이 같음)

- 트랜잭션을 지원하는 쓰기 지연(Transactional write-behind)

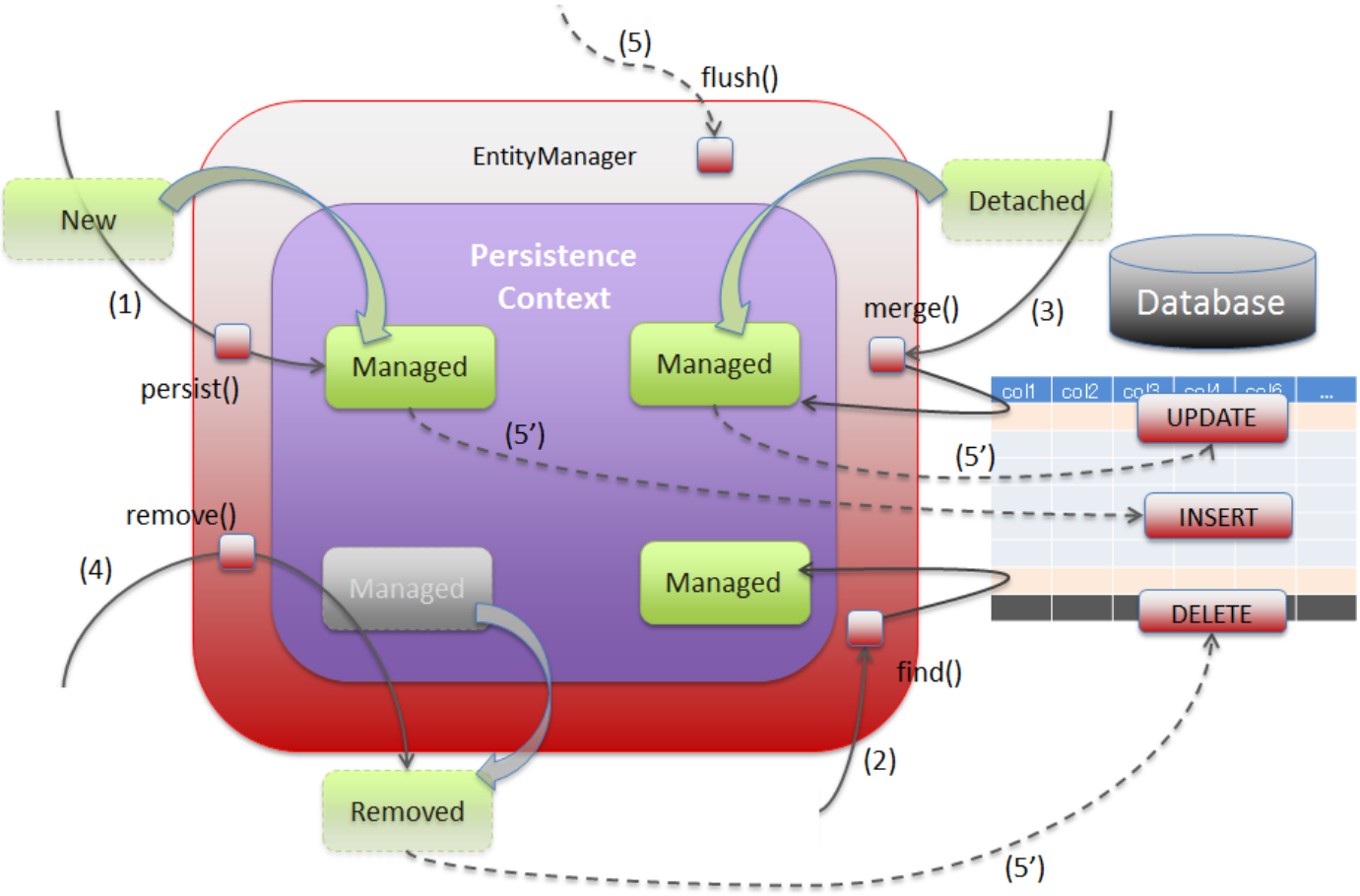
트랜젝션 커밋 될때까지 내부 쿼리저장소에 모아뒀다가 한 번에 실행

- 변경감지(Dirty Checking)

엔티티의 스냅샷을 유지하면서 엔티티의 변경사항을 체크한다. update 쿼리가 항상 같음.

- 지연로딩(Lazy Loading)

연관된 엔티티를 모두 불러오는 것이 아니라, 실제 호출될 때 로딩되도록 지원(프록시 객체 사용)



① EntityManager의 persist 메서드를 호출하면 인수로 전달된 Entity(생성 상태의 Entity)가 관리 상태의 Entity로 영속성 컨텍스트(PersistenceContext)에 저장된다.

② EntityManager의 find 메서드를 호출하면 인수로 전달된 ID를 가진 관리 상태의 Entity가 반환된다. 영속성 컨텍스트(PersistenceContext)에 존재하지 않는 경우는 Query를 실행하여 관계형 데이터베이스로부터 매핑할 레코드를 검색하고 관리 상태의 Entity로 저장한다.

③ EntityManager의 merge 메서드를 호출하면 인수로 전달된 Entity(분리 상태의 Entity)의 상태가 관리 상태의 Entity에 병합된다. 영속성 컨텍스트(PersistenceContext)에 존재하지 않는 경우는 Query를 생성하여 관계형 데이터베이스로부터 매핑 대상의 레코드를 검색하고 관리 상태의 Entity로 저장한 후 인수로 전달된 Entity의 상태와 병합된다. 이 메서드를 호출하면 persist 메서드와 달리 인수로 전달된 Entity가 관리 상태의 Entity로 저장되는 것은 아니다.

④ EntityManager의 remove 메서드를 호출하면 인수로 전달된 관리 상태의 Entity가 삭제 상태의 Entity가 된다. 이 메서드를 호출하면 삭제 상태가 된 Entity를 얻을 수 없게 된다.

⑤ EntityManager의 flush 메서드를 호출하면 persist(), merge(), remove() 메서드에 의해 축적된 Entity에 대한작업이 관계형 데이터베이스에 반영된다. flush() 메서드를 호출하는 것으로 Entity에 대한 변경 내용은 관계형 데이터베이스의 레코드에 동기화된다. EntityManager의 find 메서드를 사용하지 않고 Query를 발행하고 Entity를 검색하면 검색 작업을 수행하기 전에 EntityManager 내부의 처리로 flush 메서드와 동일한 처리가 실행되고 축적된 Entity에 대한 작업이 관계형 데이터베이스에 반영된다.

3. JPA 객체지향 쿼리 언어

JPA는 다양한 쿼리 방법을 지원한다.

- JPQL
- JPA Criteria
- QueryDSL
- 네이티브 SQL

[JPQL(Java Persistence Query Language)]

SQL을 추상화한 객체 지향 쿼리 언어이다.
SQL과 문법이 유사함. (SELECT, FROM, WHERE, GROUP BY, HAVING, JOIN 지원)
JPQL은 엔티티 객체를 대상으로 쿼리를 수행함.

(SQL은 데이터베이스 테이블을 대상으로 쿼리를 수행함)
SQL을 추상화 했기에 특정 DB 문법에 의존하지 않음.

List<Member> result = em.createQuery("select m from Member m where m.username like '%kim%'", Member.class).getResultList();

[JPA Criteria]
- JPA 공식 기능이며 JPQL 빌더 역할을 한다.
- 자바코드로 쿼리문을 작성함.
- 자바로 작성했기에 동적쿼리로 작성하기 쉽고, 코드가 잘못되면 컴파일 오류가 나기에 쉽게 고치는
  장점이 있다.
- 하지만 너무 복잡하고 실용성이 없어서 굳이 사용한다면 이것 대신에 QueryDSL 사용을 권장한다.

[QueryDSL]
- 문자열이 아닌 자바코드로 JPQL로 작성 가능하다.
- JPQL 빌더 역할을 하며, 컴파일 시점에 문법 오류를 찾을 수 있다.
- Criteria 처럼 동적 쿼리를 작성하기 편하다.
- SQL과 비슷하여 단순하고 쉬우며 실무 사용에 권장된다.

```
//JPQL : select m from Member m where m.age > 18

JPAFactoryQuery query = new JPAQueryFactory(em);
QMember m = QMember.member;
List<Member> list = query.selectFrom(m)
                         .where(m.age.gt(18))
                         .orderBy(m.name.desc())
                         .fetch();
```

[네이티브 SQL]

- JPA가 제공하는 SQL을 직접 사용하는 기능이다.
- JPQL로 해결할 수 없는 특정 데이터베이스에 의존적인 기능이다.
  (Oracle의 CONNECT BY, 특정 DB만 사용하는 SQL 힌트 등)
- EntityManager 객체의 createNativeQuery() 를 사용한다.

[JDBC 직접 사용, SpringJdbcTemplate 등]

- JPA로 JDBC 커넥션을 직접 사용하거나 SpringJdbcTemplate, MyBatis 등을 함께 사용 가능하다.
- 주의할 점은 적절한 시점에 영속성 컨텍스트에 대한 강제 flush가 필요하다.

- JPQL 구문

1. JPQL이란?

엔티티 객체를 조회하는 객체지향 쿼리이다.

문법은 SQL과 비슷하고 ANSI 표준 SQL이 제공하는 기능을 유사하게 지원한다.

SQL은 데이터베이스 테이블을 대상으로 JPQL은 엔티티 객체를 대상으로 쿼리한다.

JPQL은 SQL을 추상화해서 특정 데이터베이스에 의존하지 않는다.

JPQL은 실행 시 SQL로 변환된다.

2. JPQL 기본 문법

select_문 :: =

select_절
from_절
[where_절]
[groupby_절]
[having_절]
[orderby_절]

(1) Query 문

select m from Member as m where m.username = 'Hello'

대소문자 구분

엔티티와 속성은 대소문자를 구분하고 JPQL 키워드는 대소문자를 구분하지 않는다.

JPQL에서 사용한 Member는 테이블 명, 클래스 명이 아니라 엔티티 명이다.

엔티티 명은 @Entity(name = "")로 지정할 수도 있다.

[별칭]

"Member as m"을 보면 Member에 m이라는 별칭을 사용했다.

JPQL은 별칭이 필수이다. (as는 생략 가능)

(2) 집합

GROUP BY : 통계 데이터를 구할 때 특정 그룹끼리 묶어준다.

HAVING : GROUP BY와 함께 사용하는데 GROUP BY로 그룹화한 통계 데이터를 기준으로 필터

링한다.

groupby_절 :: = GROUP BY {단일값 경로 | 별칭}

having_절 :: = HAVING 조건식

```
select t.name from Member m LEFT JOIN m.team t
        GROUP BY t.name Having AVG(m.age) >= 10
```

- 집합 함수

COUNT	결과 수를 구한다. 반환 타입 : Long
MAX, MIN	최대, 최소 값을 구한다. 문자, 숫자, 날짜 등에 사용한다.
AVG	평균값을 구하며 숫자타입만 사용할 수 있다. 반환 타입 : Double

SUM 합을 구하며 숫자타입만 사용할 수 있다.

NULL 값은 무시하므로 통계에 잡히지 않는다.

DISTINCT를 집합 함수 안에 사용해서 중복된 값을 제거하고 나서 집합을 구할 수 있다.

select COUNT(DISTINCT m.age) from Member m

(3) 정렬

ORDER BY : 결과를 정렬할 때 사용한다.
ASC : 오름차순(기본값)
DESC : 내림차순

orderby_절 :: = ORDER BY { 상태필드 경로 | 결과 변수 [ASC | DESC] }

select m from Member m order by m.age DESC, m.username ASC

(4) 서브 쿼리

JPQL도 SQL처럼 서브 쿼리를 지원한다.
서브 쿼리를 WHERE, HAVING 절에서만 사용할 수 있고 SELECT, FROM 절에서는 사용할 수
없다.

select m from Member m where m.age > (select avg(m2.age) from Member m2)

- EXISTS

문법 : [NOT] EXISTS (subquery)

설명 : 서브 쿼리에 결과가 존재하면 참이다. NOT은 반대

//teamA 소속인 회원

select m from Member m where exists (select t from m.team t where t.name = 'teamA')

- {ALL | ANY | SOME}

문법 : {ALL | ANY | SOME} (subquery)

설명 : 비교 연산자와 같이 사용한다. { = | > | >= | < | <= | <> }

All : 조건을 모두 만족하면 참이다.

ANY or SOME : 둘은 같은 의미다. 조건을 하나라도 만족하면 참이다.

```
//전체 상품 각각의 재고보다 주문량이 많은 주문들
select o from Order o where o.orderAmount > ALL
                (select p.stockAmount from Product p)
```

- [NOT] IN

문법 : [NOT] IN (subqeury)

설명 : 서브 쿼리의 결과 중 하나라도 같은 것이 있으면 참이다.

IN은 서브 쿼리가 아닌 곳에서도 사용한다.

```
//20세 이상을 보유한 팀
select t from Team t where t IN
        (select t2 from Team t2 join t2.members m2 where m2.age >= 20)
```

(5) 쿼리 객체

작성한 JPQL을 실행하려면 쿼리 객체를 만들어야 한다.

쿼리 객체는 TypeQuery와 Query 두 가지가 있다.

- TypeQuery

반환할 타입을 명확하게 지정할 수 있을 때 사용한다.

```
TypedQuery<Member> query = em.createQuery("select m from Member as m",
                Member.class);
List<Member> members = query.getResultList();
```

```
TypedQuery<String> query = em.createQuery("select m.username from Member as m",
                String.class);
List<String> names = query.getResultList();

TypedQuery<Long> query = em.createQuery(
                "select count(m.username) from Member as m", Long.class);
Long su = query.getSingleResult();
```

- Query

반환 타입을 명확하게 지정할 수 없을 때 사용한다.

```
Query query = em.createQuery("select m.username, m.age from Member m");
```

조회 타입이 String 타입의 username과 Integer 타입의 age이므로 반환 타입이 명확하지 않다.

Query객체는 조회 대상이 둘 이상이면 Object[]를 반환하고 조회 대상이 하나면 Object를 반환
한다.

```
List<Object[]> resultList = query.getResultList();

for (Object[] objects : resultList) {
    String userName = (String) objects[0];
    Integer age = (Integer) objects[1];
  }

List resultList = query.getResultList();
for (Object o : resultList) {
```

```
            Object[] objects = (Object[]) o;
            String userName = (String) objects[0];
            Integer age = (Integer) objects[1];
        }

        Query query = em.createQuery("select count(m.username) from Member m");
        Object su = query.getSingleResult();

        Query query = em.createQuery("select m.username from Member m");
        List<Object> resultList = query.getResultList();
```

(6) 결과 조회 API

- query.getResultList()

결과가 하나 이상 일 때 또는 몇 개일지 예측할 수 없을 때 사용
결과가 없으면 빈 리스트 반환

```
        TypedQuery<Member> query = em.createQuery("select m from Member as m",
                                                  Member.class);
        List<Member> resultList = query.getResultList();
```

- query.getSingleResult()

결과가 정확히 하나, 단일 객체 반환
결과가 없으면 : javax.persistence.NoResultException
둘 이상이면 : javax.persistence.NonUniqueResultException

```
TypedQuery<Member> query = em.createQuery(
                "select m from Member as m where m.id = 1L", Member.class);
Member result = query.getSingleResult();
```

(7) 파라미터 바인딩

JPQL은 이름 기준, 위치 기준 2가지 파라미터 바인딩을 지원한다.

- 이름 기준

파라미터를 이름으로 구분하는 방법이다.

이름 기준 파라미터는 앞에 :를 사용한다.

```
Member result = em.createQuery(
        "select m from Member as m where m.username= :username",
Member.class)
        .setParameter("username", "member1")
        .getSingleResult();
```

:username이라는 이름 기준 파라미터를 정의한다.

query.setParameter() 에서 username이라는 이름으로 파라미터를 바인딩한다.

- 위치 기준

파라미터를 위치로 구분하는 방법이다.

? 다음에 위치 값을 주면 된다. (위치 값은 1부터 시작한다.)

```
List<Member> members = em.createQuery(
        "select m from Member m where m.username = ?1", Member.class)
        .setParameter(1, "member1")
        .getResultList();
```

위치 기준 파라미터 방식보다는 이름 기준 파라미터 바인딩 방식을 사용하는 것이 더 명확하다.

(8) 프로젝션(Projection)

SELECT 절에 조회할 대상을 지정하는 것을 프로젝션(projection)이라 한다.

- 엔티티 프로젝션

원하는 객체를 조회한다고 생각할 수 있다.

조회한 엔티티는 영속성 컨텍스트에서 관리된다.

select m from Member m

select m.team from Member m

- 스칼라 타입 프로젝션

숫자, 문자, 날짜와 같은 기본 데이터 타입들을 스칼라 타입이라 한다.

```
List<String> result = em.createQuery("select m.username from Member m",
                        String.class).getResultList(); //전체 회원의 이름을 조회
```

- 여러 값 조회

원하는 데이터들만 선택해서 조회할 수 있다.

```
List<Object[]> resultList = em.createQuery(
        "select m.username, m.age from Member as m").getResultList();
for (Object[] result : resultList) {
    String username = (String) result[0];
    Integer age = (Integer) result[1];
}
```

(9) 페이징 API

JPA는 페이징을 다음 두 API로 추상화 하였다.

setFirstResult (int startPosition) : 조회 시작 위치 (0부터 시작)
setMaxResult (int maxResult) : 조회할 데이터 수

```
List<Member> result = em.createQuery(
                "select m from Member m order by m.age desc", Member.class)
                .setFirstResult(1)
                .setMaxResults(10)
                .getResultList();
```

시작 index 1부터 총 10개의 데이터를 조회한다.
데이터베이스마다 다른 페이징 처리를 같은 API로 처리할 수 있게 되었다.
MYSQL, ORACLE, SQLServer 등의 dialect(방언)을 고려해서 JPA가 처리해준다.

6. JPA 기본키 매핑

기본키(primary key)를 매핑하는 방법은 2가지로 직접 할당과 자동 생성이 있다.
직접 할당은 엔티티에 @Id 어노테이션만 사용해서 직접 할당하는 것이다.
자동 생성은 엔티티에 @Id와 @GeneratiedValue를 추가하고 원하는 키 생성 전략을 선택한다.
자동 생성같은 경우에는 MySQL의 AUTO_INCREMENT 같은 기능으로 생성된 값을 기본키로 사용하는 것이다.

IDENTITY : 기본키 생성을 DB에 위임한다.
SEQUENCE : DB 시퀀스를 사용해서 기본키를 할당한다.
TABLE : 키 생성 테이블을 사용한다.
AUTO : 선택한 DB에 따라 IDENTITY, SEQUENCE, TABLE 중 하나를 자동으로 선택한다.

- 기본키 자동 생성 전략 - IDENTITY

기본키 생성을 DB에 위임하는 전략이다.
MySQL, PostgreSQL, SQL Server, DB2에서 사용한다.
(MySQL의 AUTO_INCREMENT)

JPA는 보통 트랜잭션 커밋 시점에 INSERT SQL 실행하는데 이 전략은 DB에 먼저 INSERT SQL을
실행해야 기본키를 알 수 있어서 em.persist() 시점에 즉시 INSERT SQL 실행하고 DB에서 식별
자를 조회하게 된다.

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

- 기본키 자동 생성 전략 - SEQUENCE

유일한 값을 순서대로 생성하는 특별한 데이터베이스 오브젝트이다.

이 시퀀스를 사용해서 기본키를 생성하게 된다.

시퀀스를 지원하는 Oracle, PostgreSQL, DB2, H2 Database에서 사용할 수 있다.

```
@Entity
@SequenceGenerator(
        name = "ET1_SEQ_GENERATOR",     // 식별자 생성기 이름
        sequenceName = "test1_seq",       // 데이터베이스에 등록되어 있는 시퀀스 이름
        initialValue = 1,                       // 처음 시작하는 수를 지정
        allocationSize = 1              // 시퀀스 한 번 호출에 증가하는 수
)
public class EntityTest {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
                    "ET1_SEQ_GENERATOR")
    private int id;
```

IDENTITY 전략과 비슷하지만 내부 동작 방식은 다르다.

시퀀스 전략은 em.persist()를 호출할 때 먼저 DB 시퀀스를 사용해서 식별자를 조회한다.

그리고 조회한 식별자를 엔티티에 할당한 후에 엔티티를 영속성 컨텍스트에 저장한다.

이후 트랜잭션을 커밋해서 플러시가 일어나면 엔티티를 DB에 저장하게 된다.

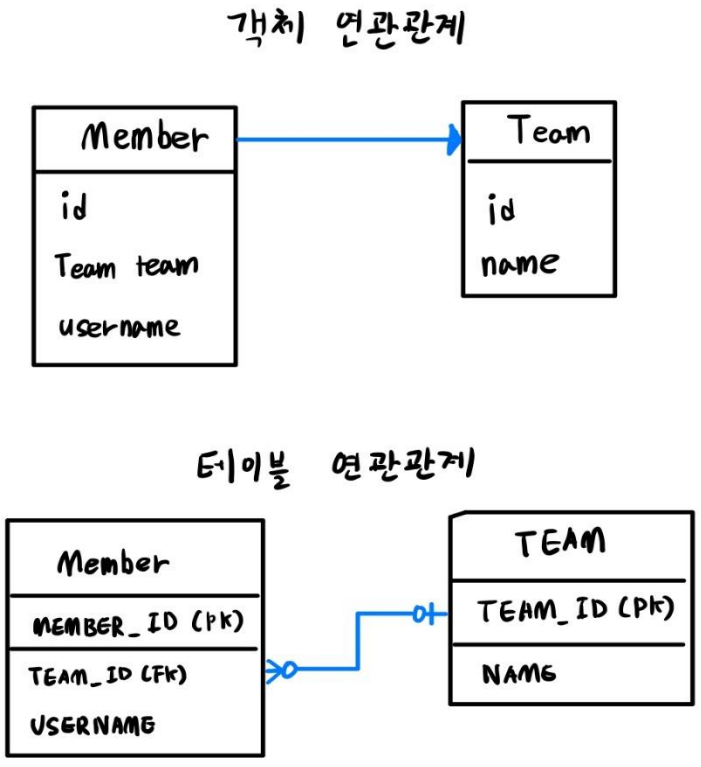
7. JPA 연관관계 매핑 기초

엔티티는 다른 엔티티와 연관관계가 대부분 존재한다.
연관관계는 단방향과 양방향이 있다.

회원 → 팀 (회원은 팀을 참조한다)
팀 → 회원 (팀은 회원을 참조한다)

위의 두 가지 관계에서 하나만 성립하는 경우 단방향 관계, 두 가지 관계 모두 서로 참조하면 양방향 관계라고 한다.

단방향 연관관계



다대일(N:1) 연관관계, 단방향

객체간의 연관관계에서는 단방향 관계이므로 member → team 조회는 가능하지만 반대의 경우는 할 수 없다. Member.team 필드를 통해서 팀을 알 수 있지만, 반대로 팀은 회원을 알 수 없다.

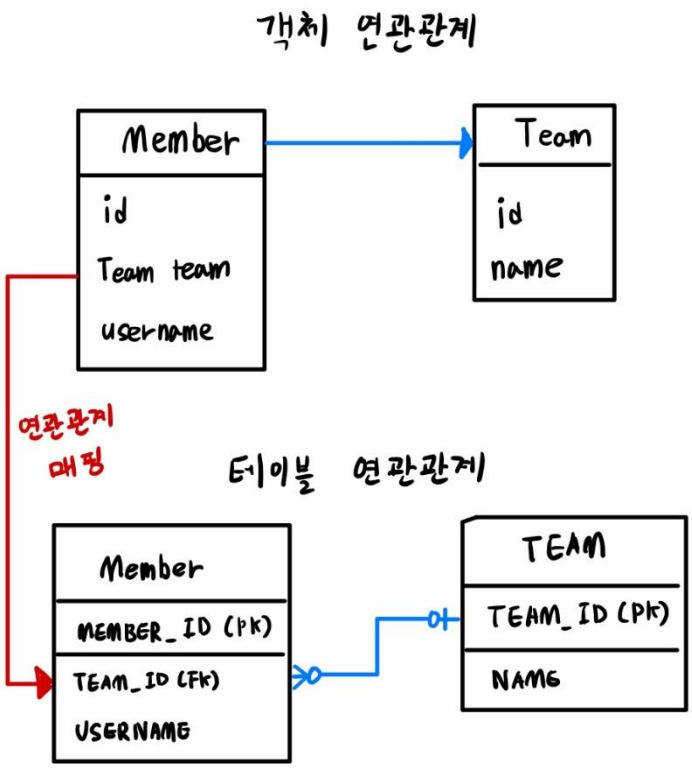
테이블 연관관계에서는 양방향 관계가 가능하므로 양쪽 조회가 가능하다.
MEMBER 테이블의 TEAM_ID 외래키로 MEMBER JOIN TEAM 과 TEAM JOIN MEMBER 둘 다 가능하다.

- 테이블은 외래키로 연관관계를 맺는다.
- 객체는 참조(주소)로 연관관계를 맺는다.

객체와 테이블의 가장 큰 차이점은 참조를 통한 객체 연관관계는 언제나 단방향이라는 것이다.
객체간에 연관관계를 양방향으로 하려면 반대쪽에도 필드를 추가해서 참조를 보관해야 한다.
즉, 단방향 관계를 반대쪽에서도 만든다는 것이다. (단방향 2개 == 양방향 1개)

- 단방향 연관관계 매핑

다대일(N:1) 연관관계, 단방향



JPA를 사용해서 객체 연관관계와 테이블 연관관계를 매핑하면 위 그림처럼 된다. 회원 객체의 Member.team 필드와 회원 테이블의 MEMBER.TEAM_ID 외래키 컬럼이 매핑되는 것이다.

- 회원 엔티티

```
@Entity
public class Member {
	@Id
	@Column(name = "MEMBER_ID")
	private String id;

	private String username;

	//연관관계 매핑
	@ManyToOne
	@JoinColumn(name = "TEAM_ID")
	private Team team;
}
```

- 팀 엔티티

```
@Entity
public class Team {
	@Id
	@Column(name = "TEAM_ID")
	private String id;

	private String name;
}
```

@ManyToOne 이란?

다대일(N:1) 관계라는 매핑 정보이다.

테이블 연관관계를 보면 회원과 팀의 관계가 N:1로 표현되어 있다.

속성	기능	기본값
optional	false로 설정하면 연관된 엔티티가 항상 있어야 한다.	TRUE
fetch	글로벌 페치 전략을 설정한다.	@ManyToOne=FetchType.EAGER @OneToMany=FetchType.LAZY
cascade	영속성 전이 기능을 사용한다.	
targetEntity	연관된 엔티티의 타입 정보를 설정한다. 이 기능은 거의 사용안함.	

@JoinColumn 이란?

@JoinColumn은 외래키를 매핑할 때 사용한다.

name 속성에는 매핑할 외래키 이름을 지정한다. 회원과 팀 테이블은 TEAM_ID 외래키로 연관관계를 맺으므로 이 값을 지정한다.

생략할 수 있다. 생략한다면 외래키를 찾을 때 기본 전략을 사용하게 된다.

기본 전략 : 필드명 + _ + 참조하는 테이블의 컬럼명 (e.g. team_TEAM_ID 외래키를 사용)

속성	기능	기본값
값name	매핑할 외래키 이름	필드명 + _ + 참조하는 테이블의 기본키 컬럼명
referencedColumnName	외래키가 참조하는 대상 테이블명의 컬럼명	참조하는 테이블의 기본키 컬럼명
foreignKey(DDL)	외래키 제약조건 직접 지정. 테이블 생성할 때만 사용.	
unique, nullable, insertable, updatable, columnDefinition, table	@Column 속성과 같다.	

[persistence.xml]

```
<persistence-unit name="emptest">
   <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
   <class>jpamvcexam.model.vo.EmpVO</class>
   <class>jpamvcexam.model.vo.EmpFreqVO</class>
   <properties>
      <property name="javax.persistence.jdbc.driver"
value="com.mysql.cj.jdbc.Driver" />
      <property name="javax.persistence.jdbc.user" value="jdbctest" />
      <property name="javax.persistence.jdbc.password" value="jdbctest" />
      <property name="javax.persistence.jdbc.url" value="URL문자열" />
      <property name="hibernate.dialect"
value="org.hibernate.dialect.MySQL8Dialect" />
      <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.hbm2ddl.auto" value="none" />
   </properties>
</persistence-unit>
```

create	기존 테이블 삭제 후 다시 생성 DROP + CREATE
create-drop	create와 같지만 종료시점에 테이블 DROP
update	변경분만 반영, 추가만 가능
validate	엔티티와 테이블이 정상 매핑 되는지만 확인