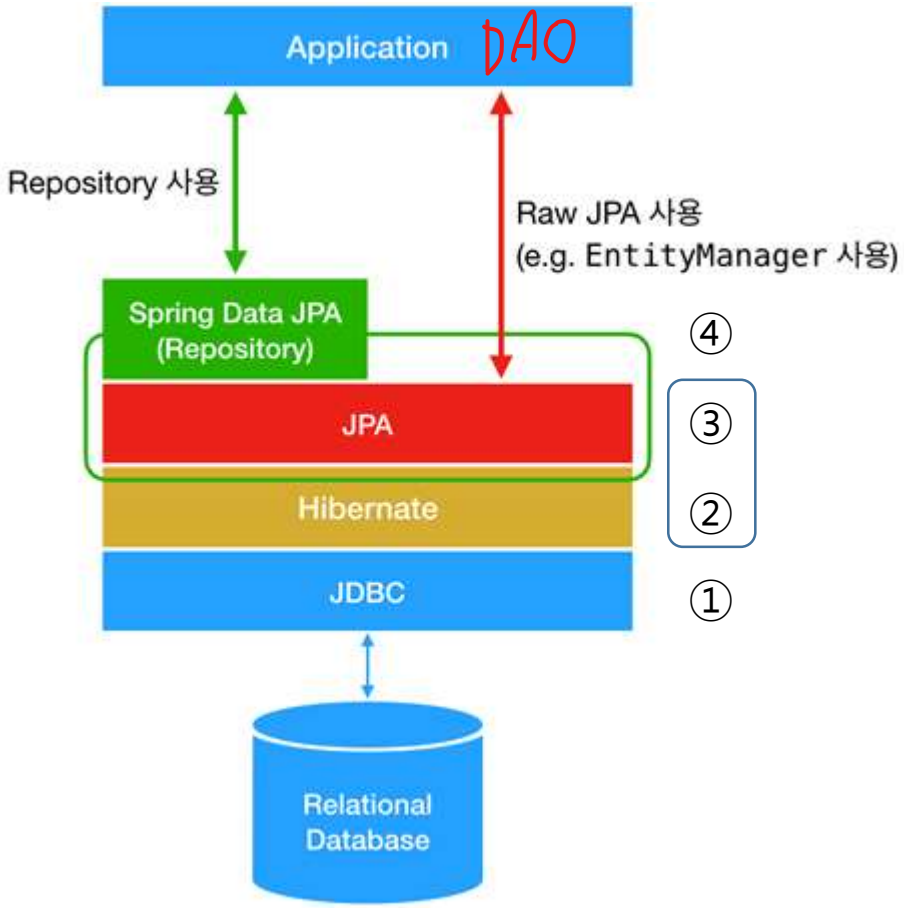
Java & Database

JPA (Java Persistence API) Overview

1. JDBC vs JPA(Hibernate) vs Spring Data JPA



1) JDBC: Low-Level Database Access

2) JPA: a technical specification that stands for Java Persistence API. It defines an interface for relational databases

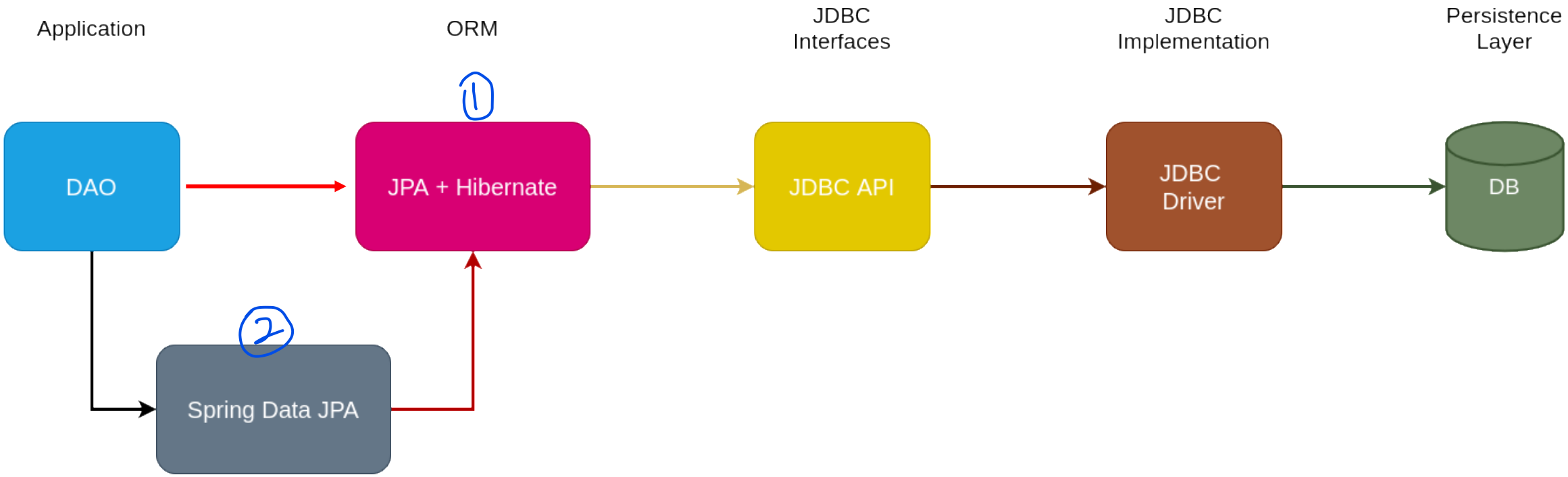
3) Hibernate: implementation of JPA

4) Spring Data JPA: a module that makes it easier to use JPA

DB 다루는 3가지 방법
(2와 3을 같이 씀)

JDBC vs JPA(Hibernate) vs Spring Data JPA

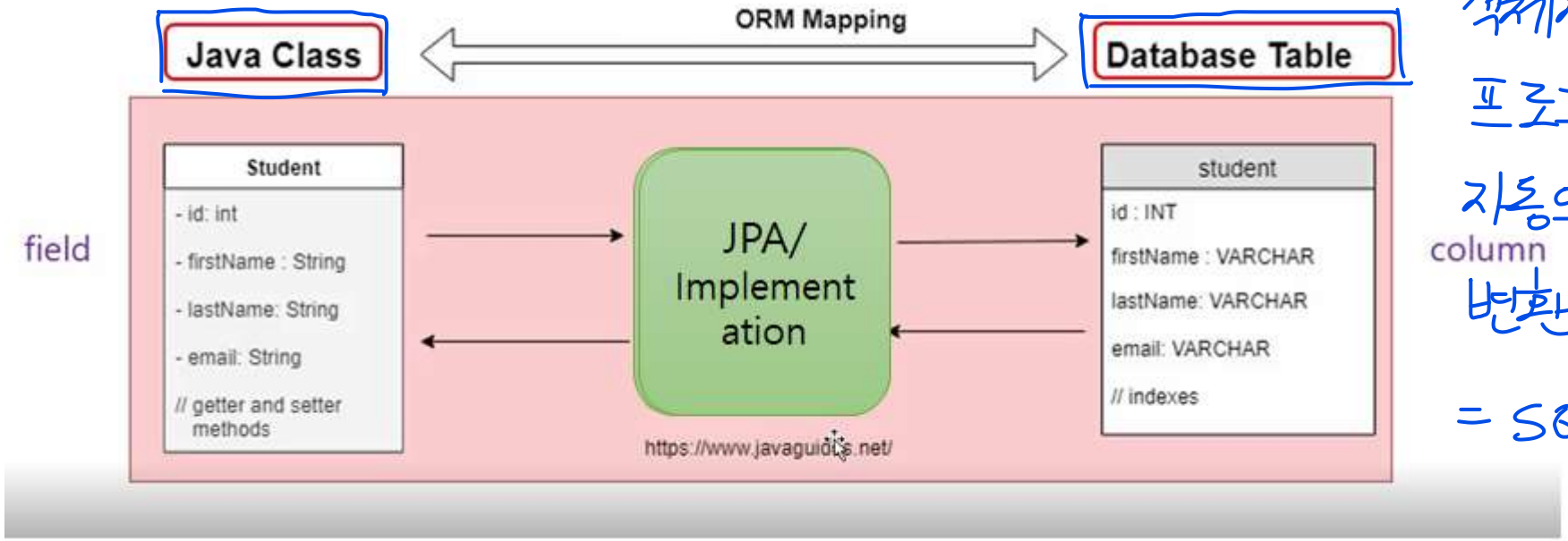
2가지를 배워볼 것이다



2 What is JPA?

JPA 란 클래스와 테이블을 매핑하는 API 다, 이 동작을 ORM 이라 한다

- Standard API for Object-to-Relational-Mapping (ORM)
  - handles all of the low-level SQL
  - help developers focus on programming with the object model



Object-To-Relational Mapping (ORM)

JPA 덕분에 객체지향적으로 프로그래밍 하더라도 자동으로 SQL로 변환해준다

= SQL 신경 쓸 필요X

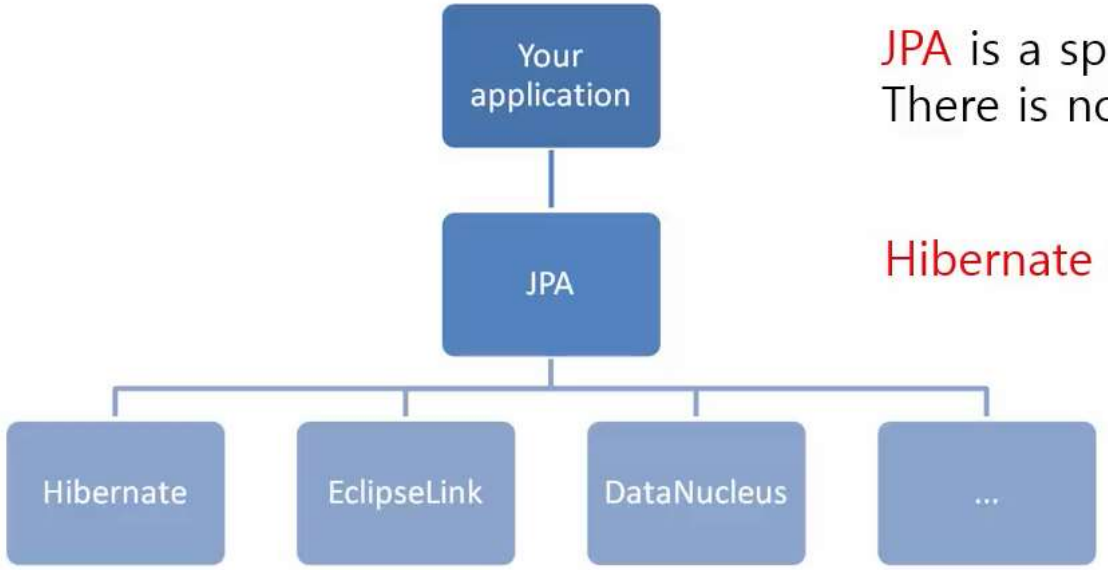
What is JPA?

JPA는 spec 이다

Hibernate기ㅡ 가장 많이 쓰인다

Hibernate가 궁극적으로 ORM 역할함

- Only a specification
  - Defines a set of interfaces
  - Requires an implementation to be usable
  - By having a standard API, you are not locked to vendor's implementation



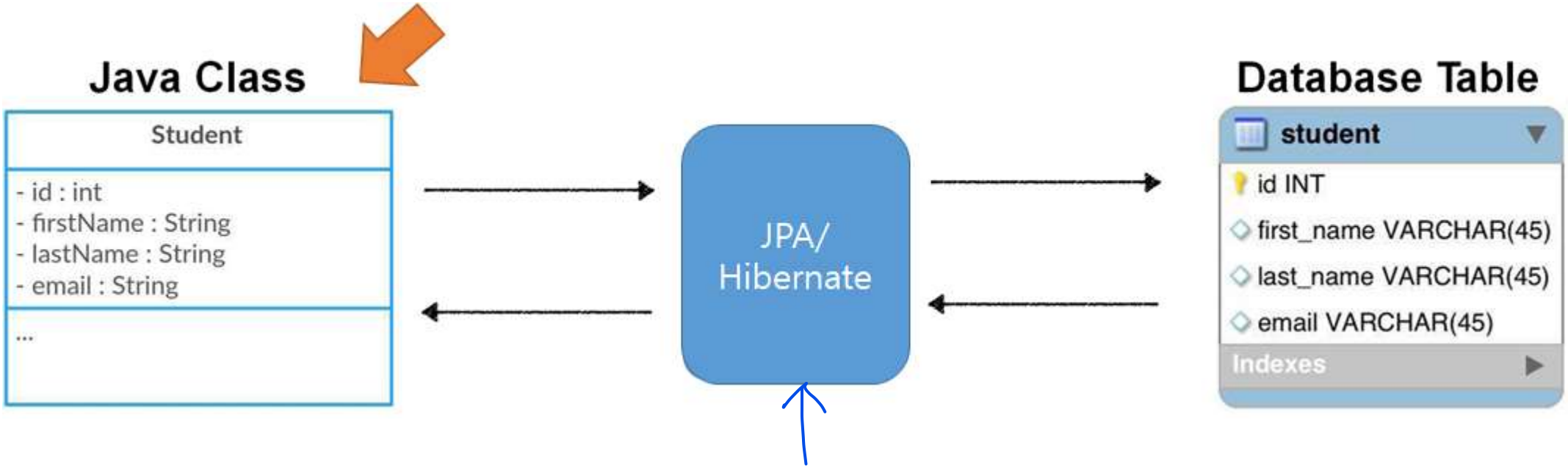
JPA is a specification or interface
There is no implementation

Hibernate is JPA implementation

2.1 Entity Class 엔티티 클래스는 자바클래스다

- Java class that is mapped to a database table

단, 테이블에 매핑되는 자바클래스 다



매핑은 JPA가 해주지만, 어떻게 매핑할 건지는 써줘야 함 (8p 아래 annotation)

Entity Class

- At a minimum, the Entity class
  - Must be annotated with @Entity 달아줘야 함
  - Must have a public or protected no-argument constructor
    - The class can have other constructors 인자없는 생성자 필수 (다른 생성자 추가 가능)
- Java Annotations
  - Step 1: Map class to database table
  - Step 2: Map fields to database columns

Entity Class

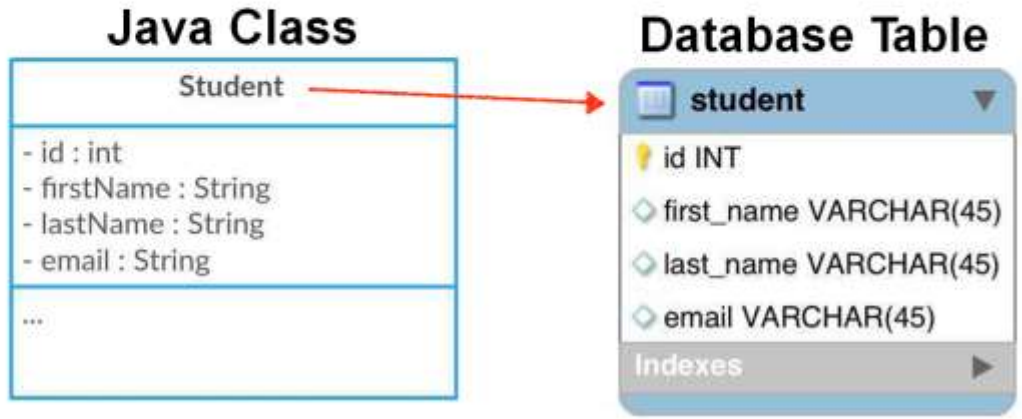
클래스는 테이블에 매핑

- Step 1: Map class to database table

```
@Entity
@Table(name="student")
public class Student {

  ...

}
```

← 테이블의 이름은 student 로 해라-

@Table 은 필수아닌 선택,
안붙이면 클래스명이 테이블명 됨



Actually, the use of @Table is optional
If not specified, database table name is same as the class

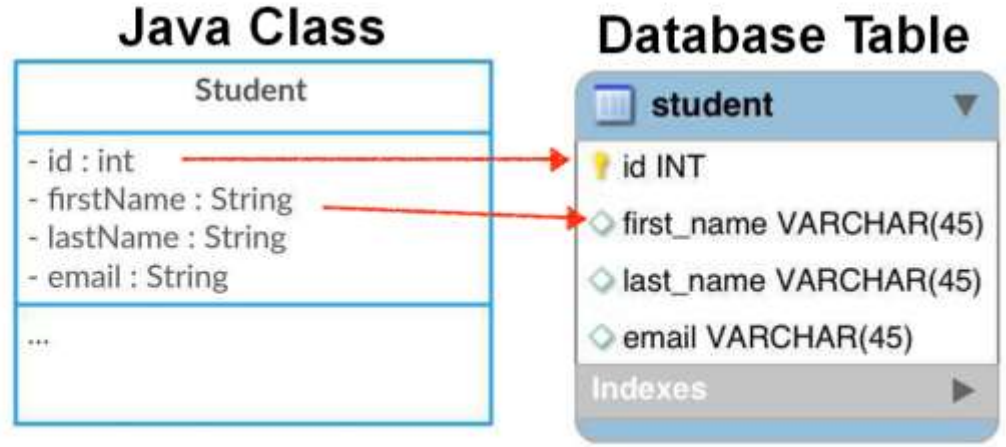
Entity Class

필드는 칼럼에 매핑

- Step 2: Map fields to database columns

```
@Entity
@Table(name="student")
public class Student {

  @Id
  @Column(name="id")
  private int id;

  @Column(name="first_name")
  private String firstName;
  ...
}
```

id가 붙어있는 칼럼은 기본키



@Column은 필수아닌 선택

Actually, the use of @Column is optional
If not specified, the column name is the same name as Java field

Entity Class

- Primary Key
  - Uniquely identifies each row in a table
  - Must be a unique value
  - Cannot contain NULL values

값을 어떻게 생성할 것인가?

```
@Entity
@Table(name="student")
public class Student {

  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  @Column(name="id")
  private int id;

  ...
}
```

MySQL

```
CREATE TABLE student (

  id int NOT NULL AUTO_INCREMENT,
  first_name varchar(45) DEFAULT NULL,
  last_name varchar(45) DEFAULT NULL,
  email varchar(45) DEFAULT NULL,
  PRIMARY KEY (id)

)
```

Entity Class

JPA의 자동생성되는 id 값을 기본키로 쓰겠다, id를 위한 테이블 안씀

ID Generation Strategies

Name	Description
GenerationType.AUTO	Pick an appropriate strategy for the particular database
GenerationType.IDENTITY	Assign primary keys using database identity column
GenerationType.SEQUENCE	Assign primary keys using a database sequence
GenerationType.TABLE	Assign primary keys using an underlying database table to ensure uniqueness

2.2 Entity Manager

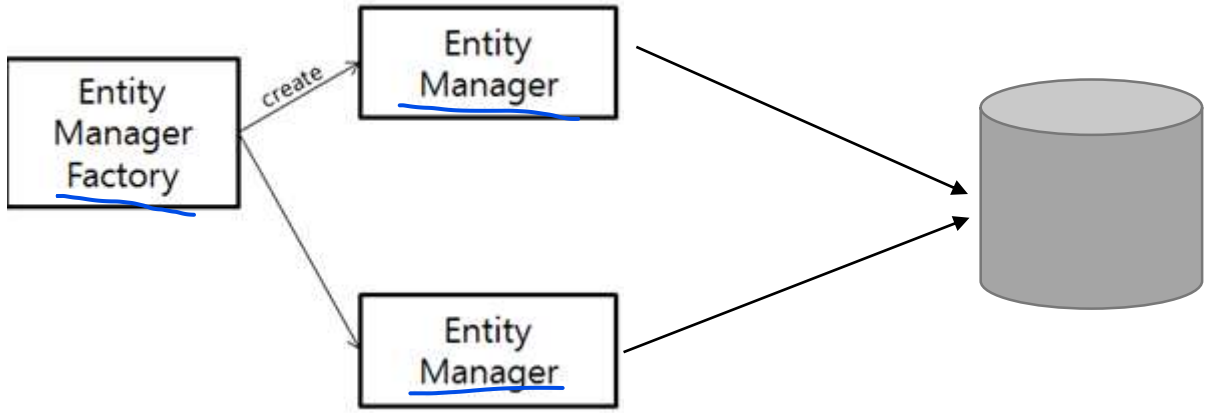
factory는 딱 한 번만 만들고,
factory가 manager를
계속 만든다

- EntityManagerFactory
  - provides *instances* of EntityManager

(트랜잭션할 때마다 만드는데,
스프링이 알아서 해주므로
우리가 신경 쓸 부분은 아님)

- JPA EntityManager
  - used to access a database in a particular application

엔티티가
테이블에 매핑된다



Entity Manager

CRUD 할 때, manager 가 제공하는 JPA 메서드들을 사용한다

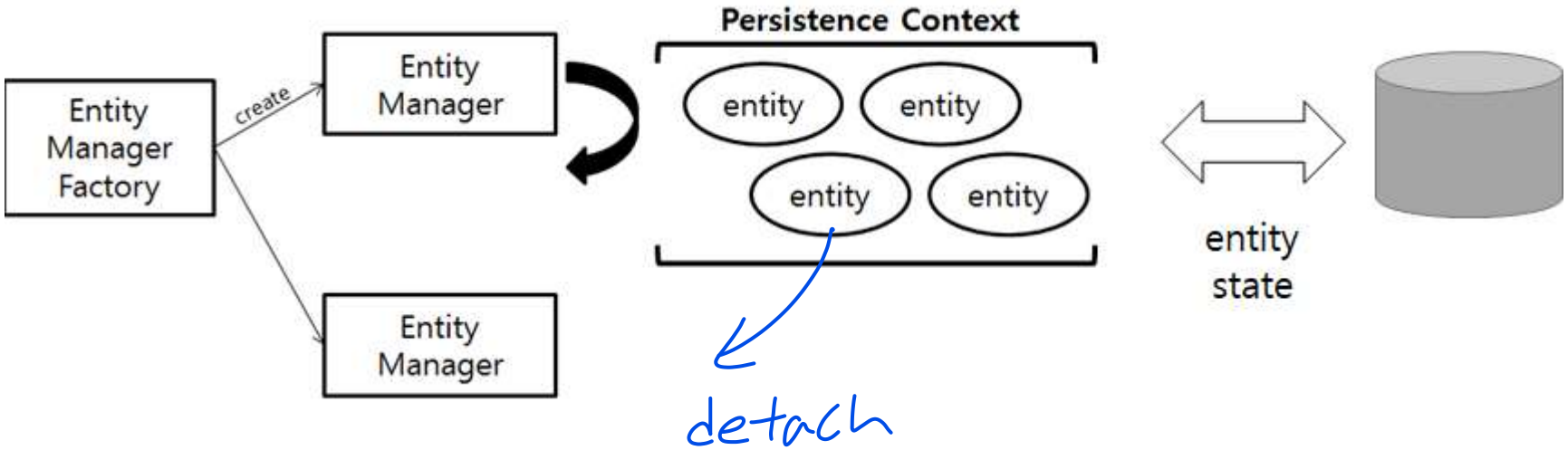
- The EntityManager API is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities

Action	JPA Methods	Hibernate Methods
Create/Save new Entity	entityManager.persist(...)	session.save(...)
Retrieve Entity by Id	entityManager.find(...)	session.get(...)/load(...)
Retrieve list of Entities	entityManager.createQuery(...)	session.createQuery(...)
Save or Update Entity	entityManager.merge(...)	session.saveOrUpdate(...)
Delete Entity	entityManager.remove(...)	session.delete(...)

2.3 Persistence Context

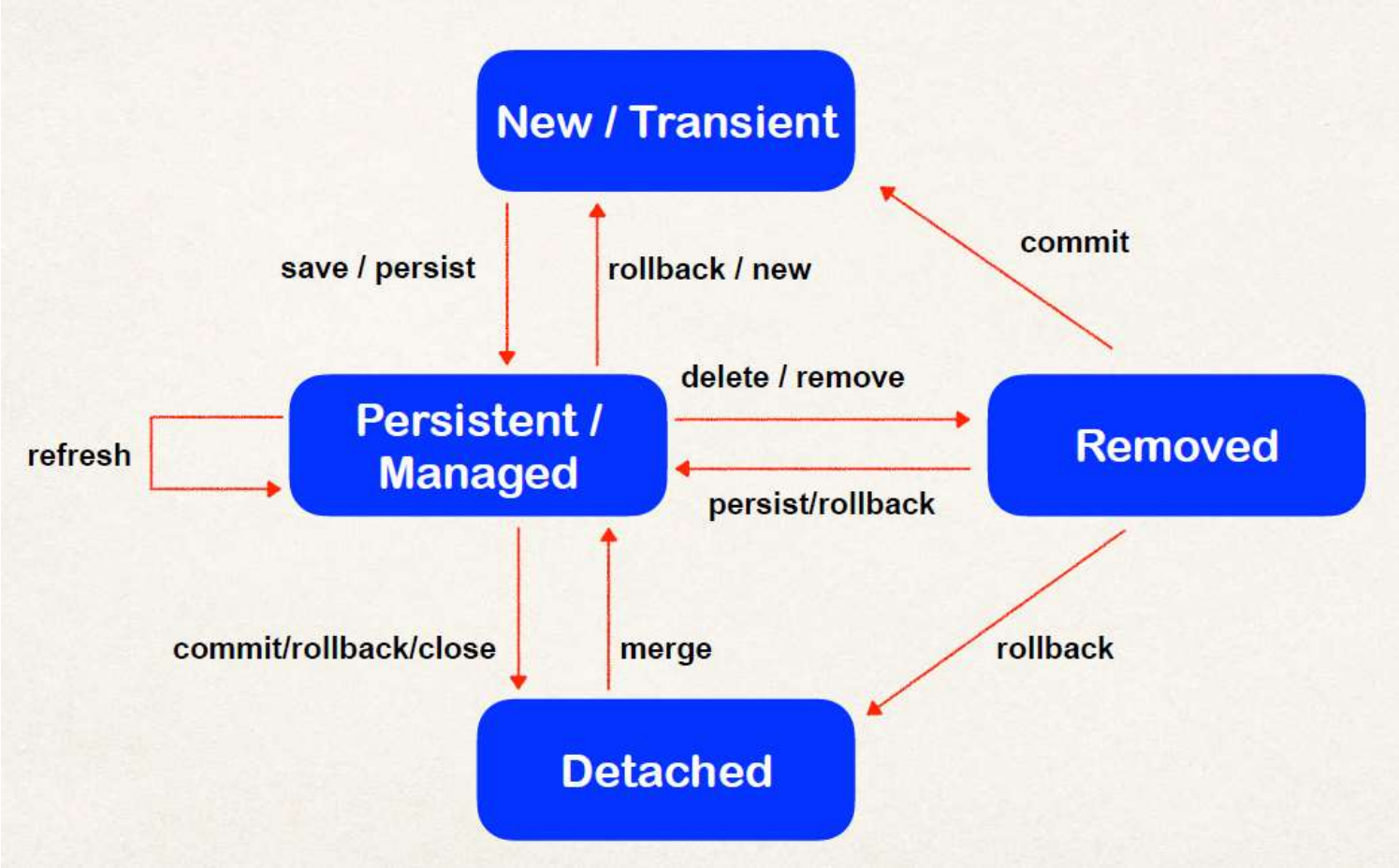
일종의 캐시 역할,
바로바로 저장하면 성능저하가
발생하므로, 캐시에 저장
해두었다가 DB에 저장하는구조

- An EntityManager instance is associated with a persistence context
- A persistence context is a set of entity instances
  - The persistence context is the first-level cache where all the entities are fetched from the database or saved to the database



DB에 저장된 엔티티 값은 id 값을 갖고있고,

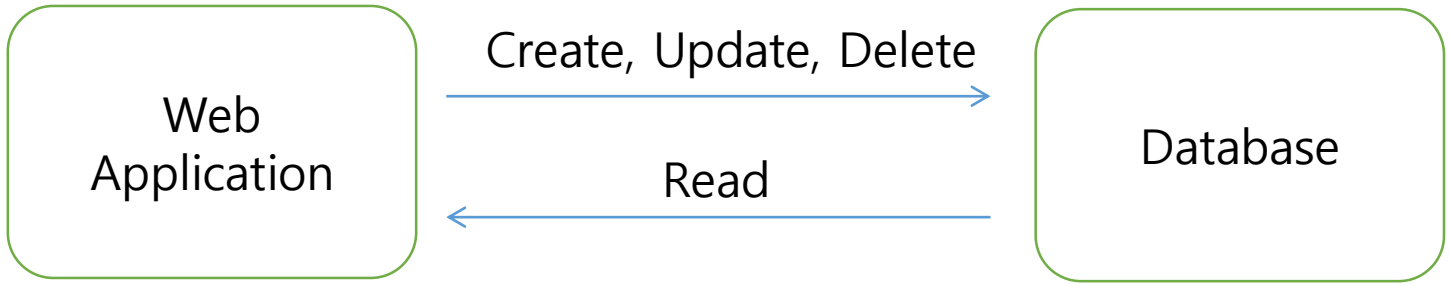
☆Entity Lifecycle 분리될 수 있다 (detach)



2.4 EntityManager API

- Create
- Read
- Update
- Delete

API를 이용하여 CRUD를 해보자



1) Saving a Java Object with JPA

DAO Class

```
@Override
public void save(Student theStudent) {
    entityManager.persist(theStudent);
}
```

객체를 DB에 저장

Save the Java object

2) Retrieving a Java Object with JPA

```
// retrieve/read from database using the primary key
// in this example, retrieve Student with primary key: 1

Student myStudent = entityManager.find(Student.class, 1);
```

조회

Entity class

Primary key

좀 더 복잡한 동작을 시키고 싶다면 SQL과 비슷한 게 있다

JPA Query Language (JPQL)

- Query language for retrieving objects
- Similar in concept to SQL
  - where, like, order by, join, in, etc...
- However, JPQL is based on entity name and entity fields

JPQL은 엔티티의 이름과 필드명이 기반한다

Retrieving all Students

테이블명 아닌 클래스명

```
TypedQuery<Student> theQuery = entityManager.createQuery("FROM Student", Student.class);

List<Student> students = theQuery.getResultList();
```

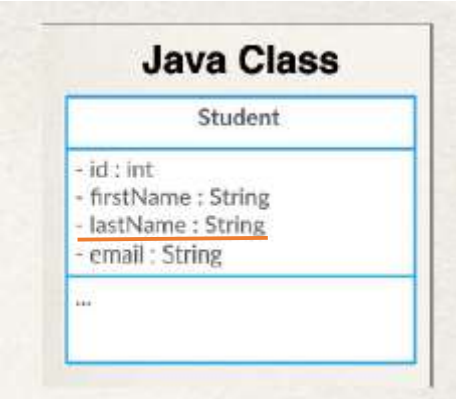
Name of JPA Entity ... the class name

Return type

Note: this is NOT the name of the database table

All JPQL syntax is based on entity name and entity fields

Retrieving Students: lastName = 'Doe'



클래스
↓
테이블 칼럼명 아닌 엔티티 필드명

Field of JPA Entity

```
TypedQuery<Student> theQuery = entityManager.createQuery(
                                "FROM Student WHERE lastName='Doe'", Student.class);

List<Student> students = theQuery.getResultList();
```

Retrieving Students using OR predicate:

엔티티 필드명

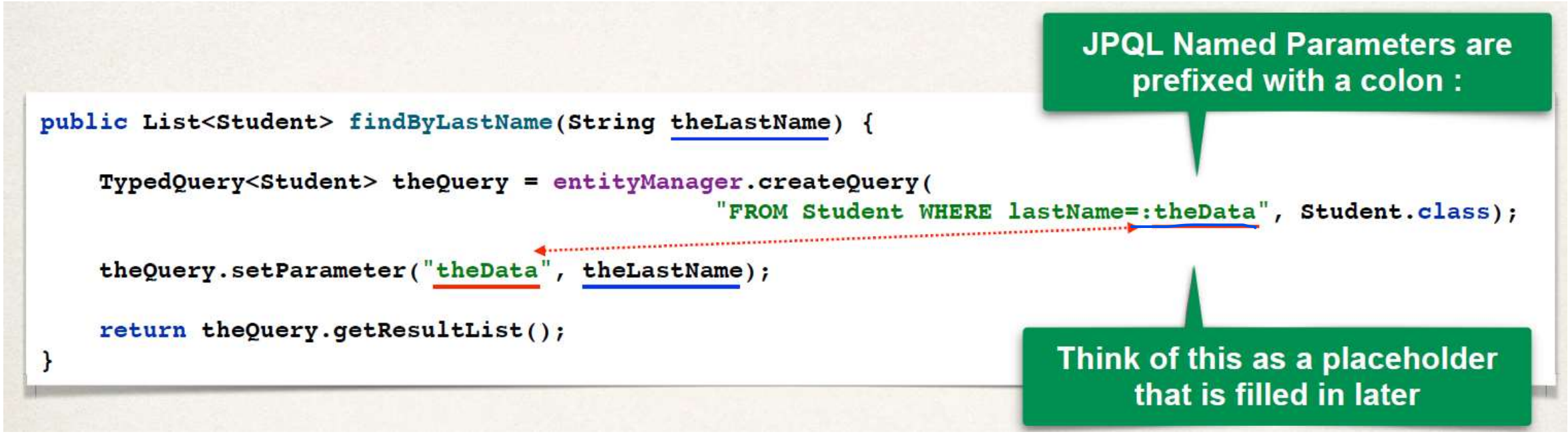
Field of JPA Entity

```
TypedQuery<Student> theQuery = entityManager.createQuery(
                    "FROM Student WHERE lastName='Doe' OR firstName='Daffy'", Student.class);

List<Student> students = theQuery.getResultList();
```

Field of JPA Entity

JPQL - Named Parameters

콜론을 포함



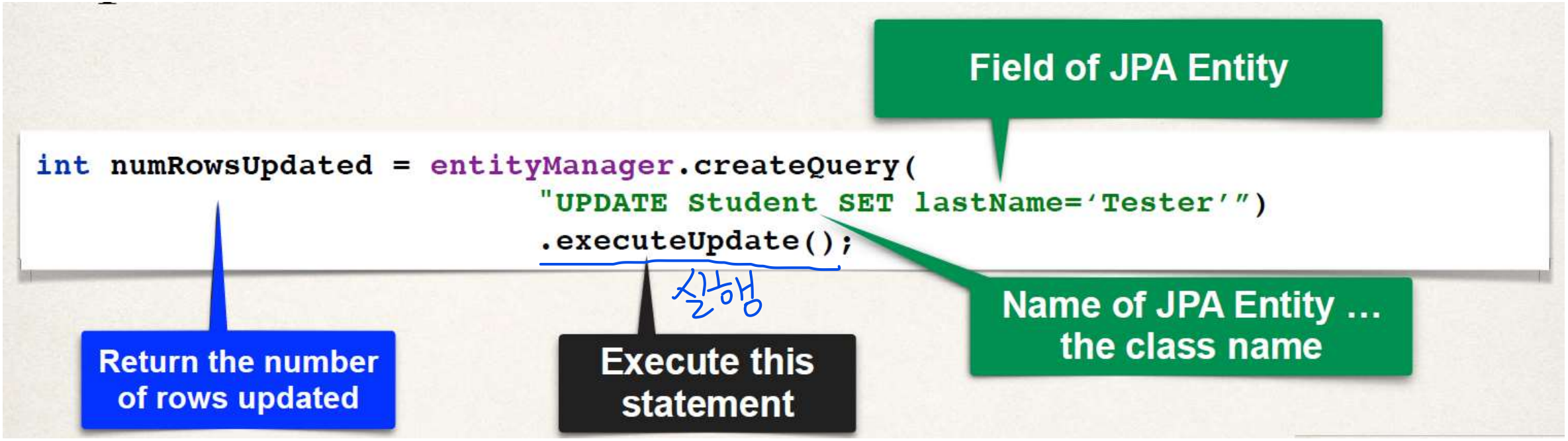
3) Update a Student

```
Student theStudent = entityManager.find(Student.class, 1);

// change first name to "Scooby"
theStudent.setFirstName("Scooby");

entityManager.merge(theStudent);
```

업데이트

Update the entity

Update last name for all students



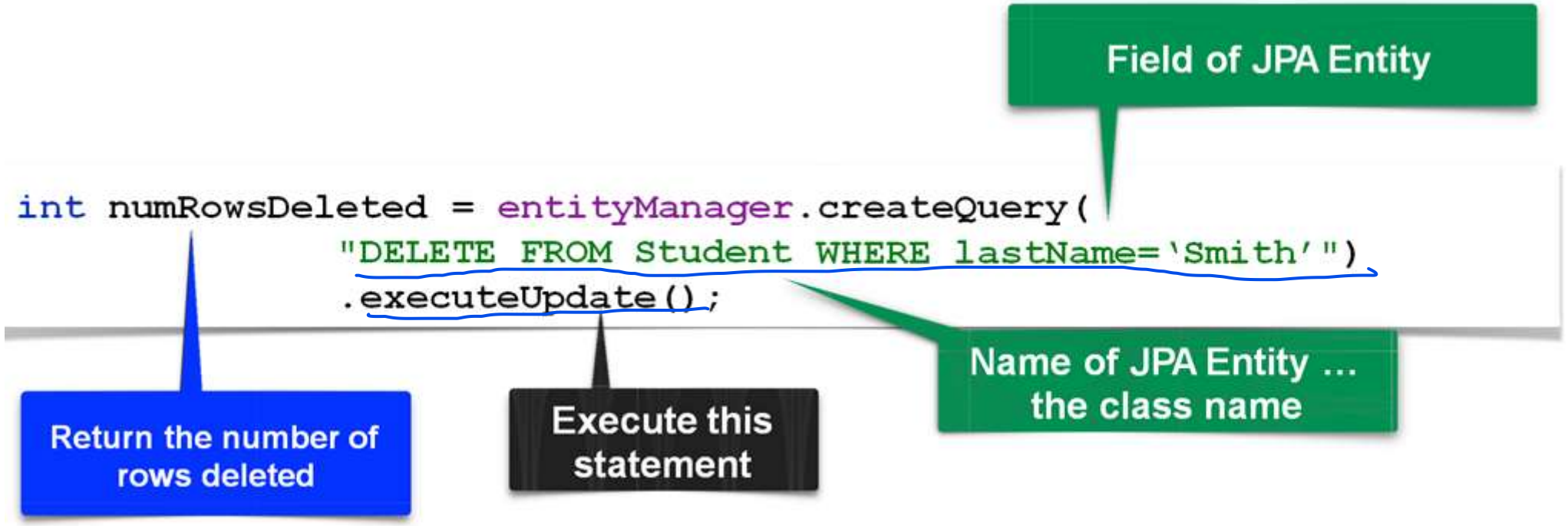
4) Delete a Student

```
// retrieve the student
int id = 1;
Student theStudent = entityManager.find(Student.class, id);

// delete the student
entityManager.remove(theStudent);
```

삭제

Delete based on a condition

조건을 걸고 DB의 삭제

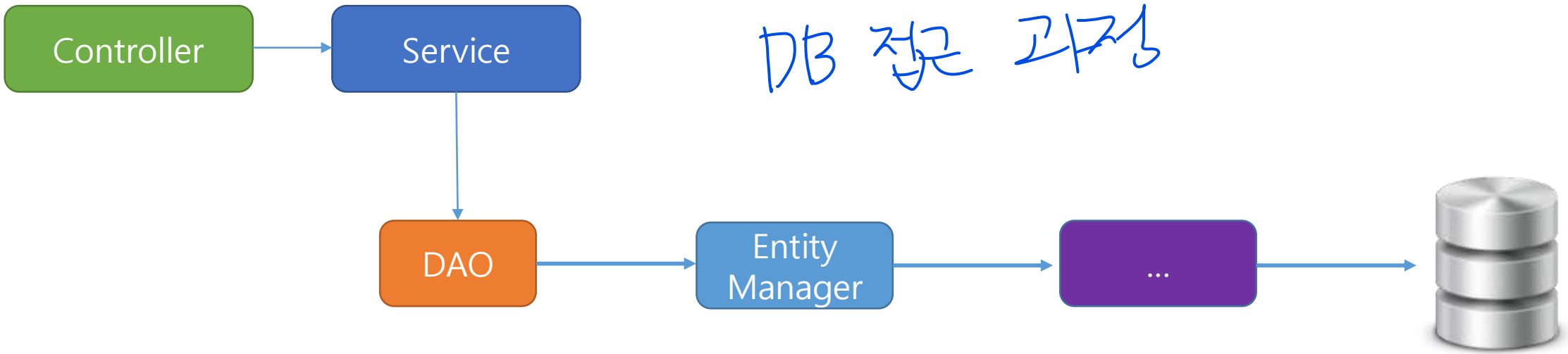


Delete All Students

```
int numRowsDeleted = entityManager
                        .createQuery("DELETE FROM Student")
                        .executeUpdate();
```

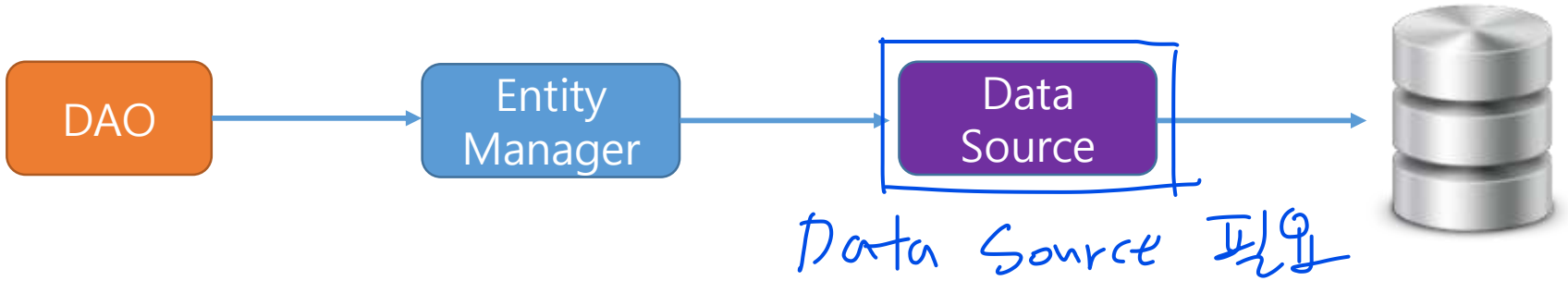
2.4 DAO(Data Access Object)

- Responsible for interfacing with the database
- need a JPA Entity Manager
  - JPA Entity Manager is the main component for saving/retrieving entities



DAO(Data Access Object)

- JPA Entity Manager
  - Our JPA Entity Manager needs a Data Source
  - The Data Source defines database connection info



  - We can autowire/inject the JPA Entity Manager into our Student DAO

```
@PersistenceContext
private EntityManager entityManager;
```

DAO(Data Access Object) 예시코드

```
@Repository
@Transactional
public class StudentDao {

  @PersistenceContext ← 의존성 주입하면 알아서 됨(?)
   private EntityManager entityManager;


   public Student findById(Integer id) {
        return entityManager.find(Student.class, id);
    }

    public List<Student> findAll() {
        // create query
        TypedQuery<Student> theQuery =
                entityManager.createQuery("FROM Student", Student.class);

        // return query results
        return theQuery.getResultList();
    }
```

```
public void save(Student theStudent) {
        entityManager.persist(theStudent);
    }
}

public void update(Student theStudent) {
        entityManager.merge(theStudent);
 }

public void delete(Integer id) {
        // retrieve the student
        Student theStudent = entityManager.find(Student.class, id);

        // delete the student
        entityManager.remove(theStudent);
}
}
```

2.5 Spring @Transactional

- Spring provides an @Transactional annotation
- Automagically begin and end a transaction for your JPA code
  - No need for you to explicitly do this in your code
- This Spring magic happens behind the scenes

```
@Repository
@Transactional
public class StudentDao {
    ...
    public void save(Student student ) {
        entityManager.persist(student);
    }

    public Student findById(Long id) {
        return entityManager.find(Student.class, id);
    }
}
```

각 메서드에 트랜잭션 시작의 끝이 자동으로 내부적으로 작성됨 → 트랜잭션 처리를 스프링이 알아서 해줌

3. Development Process

- Maven Dependency
- Configuration
- Entity Class
- DAO Class

3.1 Maven Dependency

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>${org.springframework-version}</version>
</dependency>

<dependency>
    <groupId>javax.persistence</groupId>
    <artifactId>javax.persistence-api</artifactId>
    <version>2.2</version>
</dependency>

<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.6.15.Final</version>
</dependency>
```

3.2 Configuration(dao-context.xml)

```
<bean id="entityManagerFactory"
        class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

    <property name="dataSource" ref="dataSource" />

    <property name="packagesToScan" value="kr.ac.hansung.cse.model" />

    <property name="jpaVendorAdapter">
      <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
        <property name="database" value="MYSQL" />
        <property name="databasePlatform" value="org.hibernate.dialect.MySQL8Dialect" />
      </bean>
    </property>

    <property name="jpaProperties">
      <props>
        <prop key="hibernate.show_sql">true</prop>
        <prop key="hibernate.format_sql">true</prop>
        <prop key="hibernate.hbm2ddl.auto">none</prop>
      </props>
    </property>
  </bean>
```

Packages to Scan for JPA Entities

JPA Implementation

JPA Properties

Configuration(dao-context.xml)

```
<bean id="transactionManager"
        class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="entityManagerFactory" />
 </bean>

 <tx:annotation-driven />
```

Enabling @Transactional

JPA Property: ddl-auto

Property Value	Property Description
none	No action will be performed
create-only	Database tables are only created
drop	Database tables are dropped
create	Database tables are dropped followed by database tables creation
create-drop	Database tables are dropped followed by database tables creation. On application shutdown, drop the database tables
validate	Validate the database tables schema
update	Update the database tables schema

Used during development or for testing

When database tables are dropped, all data is lost

3.3 Entity Class

```
@Entity
@Table(name = "offers")
public class Offer {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Size(min=2, max=100, message="Name must be between 2 and 100 chars")
    private String name;

    @Email(message="Please provide a valid email address")
    @NotEmpty(message="The email address cannot be empty")
    private String email;

    @Size(min=5, max=100, message="Text must be between 5 and 100 chars")
    private String text;
}
```

3.4 DAO

```
@Repository
@Transactional
public class OfferDao {

    @PersistenceContext
    private EntityManager entityManager;

    public Offer getOffer(String name) {
        return entityManager.createQuery("SELECT o FROM Offer o WHERE o.name = :name", Offer.class)
                .setParameter("name", name)
                .getSingleResult();
    }

    public Offer getOffer(int id) {
        return entityManager.find(Offer.class, id);
    }

    public List<Offer> getOffers() {
        return entityManager.createQuery("SELECT o FROM Offer o", Offer.class)
                .getResultList();
    }
```

R

DAO

```
    public void insert(Offer offer) {
        entityManager.persist(offer);
    }

    public void update(Offer offer) {
        entityManager.merge(offer);
    }

    public void delete(int id) {
        Offer offer = entityManager.find(Offer.class, id);
        entityManager.remove(offer);
    }
}
```

C

U

D

3.5 Unit Test for DAO

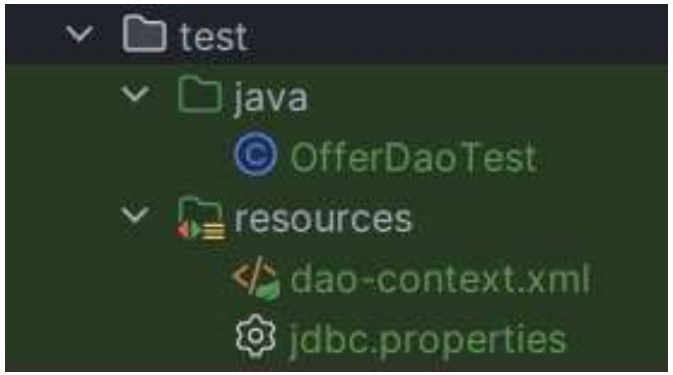
```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-engine</artifactId>
    <version>5.8.2</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>${org.springframework-version}</version>
    <scope>test</scope>
</dependency>
```

Junit4 제거

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
    <scope>test</scope>
</dependency>
```

Unit Test for DAO

src/test/java



OfferDaoTest

```
@Transactional
@ExtendWith(SpringExtension.class)
@ContextConfiguration(locations = "/dao-context.xml")

public class OfferDaoTest {

    @Autowired
    private OfferDao offerDao;

    @BeforeEach
    public void setUp() {
        Offer offer = new Offer();
        offer.setName("Test Offer");
        offer.setEmail("test@example.com");
        offer.setText("This is a test offer");

        offerDao.insert(offer);
    }
```

This is a Spring context configuration file. It is used by the Spring container to create and configure beans.

```
@Test
@DisplayName("Test1: testGetOfferByName")
    public void testGetOfferByName() {
        Offer offer = offerDao.getOffer("Test Offer");
        assertNotNull(offer);
        assertEquals("Test Offer", offer.getName());
    }
```

```
@Test
@DisplayName("Test2: testGetOfferById")
 public void testGetOfferById() {

     Offer savedOffer = offerDao.getOffer("Test Offer");

     Offer offer = offerDao.getOffer(savedOffer.getId());
      assertNotNull(offer);
      assertEquals(savedOffer.getId(), offer.getId());
}

 @Test
 @DisplayName("Test3: testGetOffers")
 public void testGetOffers() {

     List<Offer> offers = offerDao.getOffers();
     assertNotNull(offers);
     assertFalse(offers.isEmpty());
}
```

```
@Test
@DisplayName("Test4: testInsert")
public void testInsert() {

    Offer newOffer = new Offer();
    newOffer.setName("New Offer");
    newOffer.setEmail("new@example.com");
    newOffer.setText("This is a new offer");
    offerDao.insert(newOffer);
    assertNotNull(newOffer.getId());

    Offer savedOffer = offerDao.getOffer(newOffer.getId());
    assertNotNull(savedOffer);
    assertEquals("New Offer", savedOffer.getName());
}
```

```
@Test
@DisplayName("Test5: testUpdate")
public void testUpdate() {

    Offer offer = offerDao.getOffer("Test Offer");
    assertNotNull(offer);
    offer.setText("Updated text");
    offerDao.update(offer);

    Offer updatedOffer = offerDao.getOffer(offer.getId());
    assertNotNull(updatedOffer);
    assertEquals("Updated text", updatedOffer.getText());
}
```

```
@Test
@DisplayName("Test6: testDelete")
   public void testDelete() {

       Offer offer = offerDao.getOffer("Test Offer");
       assertNotNull(offer);
       offerDao.delete(offer.getId());

       Offer deletedOffer = offerDao.getOffer(offer.getId());
       assertNull(deletedOffer);
   }
```

3.6 Entity Lifecycle

```
@Entity
@Table(name="student")
public class Student {

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

    private String firstName;
    private String lastName;
    private String email;
}
```

```
@Transactional
@ExtendWith(SpringExtension.class)
@ContextConfiguration(locations = "/dao-context.xml")

public class StudentTest {

    @PersistenceContext
    private EntityManager testEntityManager;
```

```
@Test
public void testStudentLifecycle() {
    // Transient 상태의 객체 생성
    Student student = new Student();
    student.setFirstName("alice");
    student.setLastName("Kim");
    student.setEmail("alice.Kim@hansung.ac.kr");

    // Persistent 상태로 전환
    testEntityManager.persist(student);

    // ID를 사용하여 엔티티 조회
    Student persistedStudent = testEntityManager.find(Student.class, student.getId());
    assertNotNull(persistedStudent, "Persisted student should not be null");
    assertEquals("alice", persistedStudent.getFirstName(), "First name should be 'alice'");
```

```
        // 변경 감지 (Dirty checking)을 통해 엔티티 업데이트
        persistedStudent.setEmail("updated.email@hansung.ac.kr");
        testEntityManager.flush(); // 변경사항을 데이터베이스에 즉시 반영

        // 업데이트 확인
        Student updatedStudent = testEntityManager.find(Student.class, persistedStudent.getId());
        assertEquals("updated.email@hansung.ac.kr", updatedStudent.getEmail(), "Email should be updated");

        // 엔티티 분리 (Detached 상태로 전환)
        testEntityManager.detach(updatedStudent);
        updatedStudent.setEmail("another.update@hansung.ac.kr");

        // 분리된 엔티티 상태에서 변경을 시도해도 데이터베이스에 반영되지 않음을 확인
        Student detachedStudent = testEntityManager.find(Student.class, updatedStudent.getId());

        assertEquals("updated.email@hansung.ac.kr", detachedStudent.getEmail(),
                                "Email should not be updated after detachment");
    }
}
```