JPA Tutorial

Adriano Botti, Antonio Le Caldare, Francesco Merola, Giacomo Ponziani

Information Systems, University of Pisa

Fall, 2019

What is and Why JPA

JPA (Java Persistence API) is a Java specification for ORM (Object Relational Mapping). The purpose of ORM is to map the concepts from ObjectOriented Programming with concepts of relational databases to let the programmer to focus on the business logic instead of the translation between the two worlds. Once the mapping is done it is possble to work with instances of classes that are directly mapped to rows of table without performing any transformation. In this tutorial we focus on Hibernate which is one of the implementation of the JPA specification.

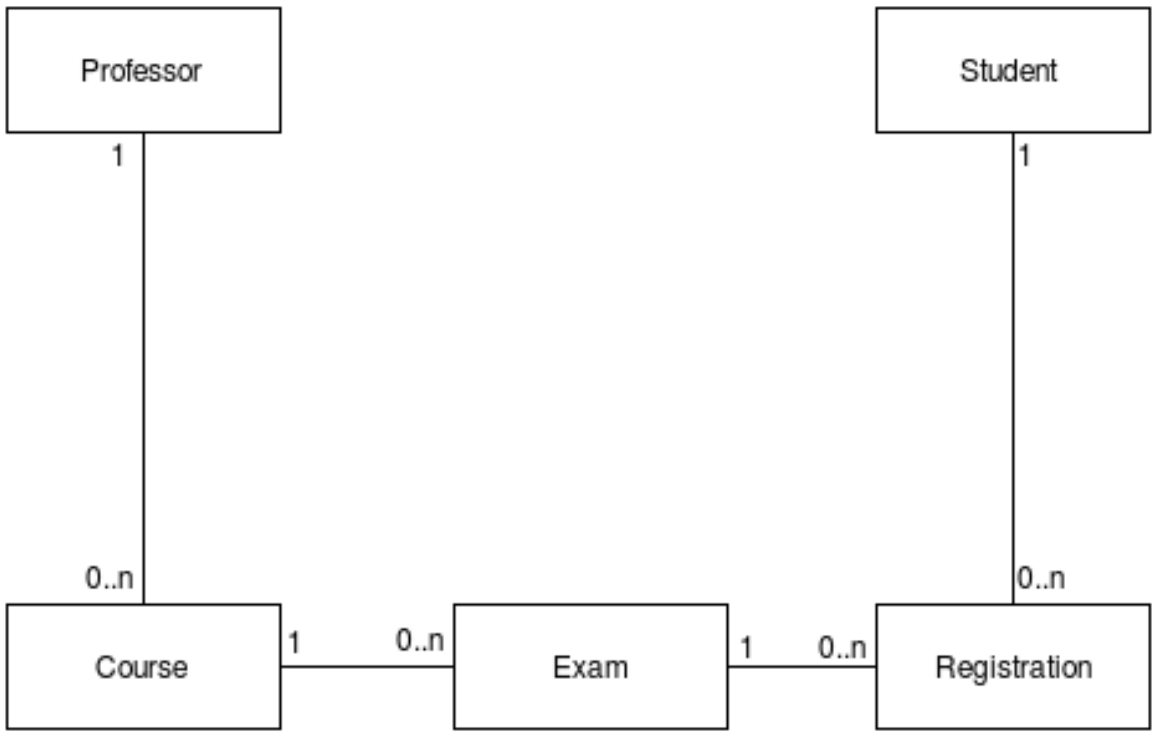
How to use JPA

Mapping may be done using *annotations* or *persistence.xml* file or both. Usually the *persistence.xml* file is used to store information about the connection with the DB while *annotations* are used actually map classes into tables.

1. Define the *persistence.xml* file;
2. Define classes using private fields with getter and setter methods and *annotations* to map them with tables;
3. Use *JPQL* to perform *CRUD* operations;

Examples Definition

All the following examples come from the application developed in *Task 1*. This application handles university exams creation, registration, deregistration and validation. For clarity the class-diagram is reported.



JPA Annotations

- *@Entity*, specifies the following class is an entity to be mapped to one or more tables in the DB.
- *@Table*, specifies the primary table for the annotated entity. The attribute *name* indicates the referenced table in the DB.
- *@Column*, specifies the mapped column for a persistent property or field. The attribute *name* indicates the name of the corresponding table column. It may be neglected when names of the attribute and of the columns are the same.

```
1 @Entity
2 @Table(name = "professor")
3 public class Professor {...}
```

Listing 1: @Entity and @Table

JPA Annotations

Annotation may be before a field or a method but not both. In the case the annotation is before the method

- *@Id*, specifies the primary key for an entity. The class of the primary key shall implement *Serializable* interface and the *equal* and *hash* methods.

```
1     @Column(name = "id")
2     @Id
3     @GeneratedValue(strategy = GenerationType.IDENTITY
      )
4     public int getId() {
5     return id;
6     }
7
```

Listing 2: @Id @Column and @GeneratedValue

JPA ANNOTATIONS

- *@Embeddable*, before a class definition indicates that the class can be included into another one tagged with *@Entity*. The Embeddable class cannot be mapped to a db table alone as it represents only a portion of it.

```
1  @Embeddable
2  public static class RegistrationId implements
       Serializable {
3    private int student;
4    private Exam.ExamID exam;
5
6    public RegistrationId() {...}
7    public int getStudent() {...}
8    public void setStudent(int student) {...}
9    public Exam.ExamID getExam() {...}
10   public void setExam(Exam.ExamID exam) {...}
11
12   @Override
13   public boolean equals(Object obj) {...}
14   @Override
15   public int hashCode() {...}
16 }
```

JPA ANNOTATIONS

- *@EmbeddedId*, indicates that the primary key of the class is composite and that it is an *@Embeddable* class. In this case the get method specifies that *id* is the primary key.

```
1  @Entity
2  @Table(name = "exam_result")
3  public class Registration {
4    private RegistrationId id;
5    \*...*\
6    @EmbeddedId
7    public RegistrationId getId() {...}
8    \*...*\
9  }
10
```

Listing 4: @EmbeddedId

JPA ANNOTATIONS

- *@ManyToOne* specifies a many to one relationship with another class. The target entity is inferred from the type of the object being referenced. The target is expeted to have a *OneToMany* annotation only if the application logic requires it, indeed in this application the *Student* class does not have such annotation since it is required to get the students data from registration and not viceversa.
- *@MapsId* states the next *@ManyToOne* (it can also be *@OneToOne*) relationship maps an attribute within the *EmbeddedId* with an attribute of another class specified with the get method: in this case *student* attribute of the embedded class is map to the *id* attribute of the *Student* class.

```
1 @MapsId("student")
2 @JoinColumn(name="student", referencedColumnName="id")
3 @ManyToOne
4 public Student getStudent () {...}
```

Listing 5: Foreign key

JPA ANNOTATIONS

- *@JoinColumn* specifies a column for joining.

```
@MapsId("exam")
@JoinColumns({
  @JoinColumn(name="course", referencedColumnName="
    course"),
  @JoinColumn(name="date", referencedColumnName="date"
    )
})
@ManyToOne
public Exam getExam() {...}
public void setExam(Exam exam) {...}
```

Listing 6: Foreign key

This is another example of foreign key: the attribute *exam* of the *Embeddable* class *RegistrationId* is mapped to the *Exam* class thorough the attributes *course* and *date*.

JPA ANNOTATIONS

- @ManyToMany, specifies a many-valued association with many-to-many multiplicity. If for example we change our application and we suppose that more than one professor can teach a course we have a many to many relationship between the two entities. We suppose that the courses are more used in our application and therefore makes sense to set *Course* entity as the *owning side* using the @JoinTable: it specifies the name of the joining table and joining columns.

JPA ANNOTATIONS

```
1  @Entity
2  @Table(name="Professor")
3  public class Professor {
4    /*
5     *
6     */
7     private Collection<Course> courses = new Collection
       <>();
8     @ManyToMany(mappedBy=professors)
9     //mappedBy indicates the attribute in the owning
       side
10    public Collection<Course> getCourses(){ return
       courses; }
11    /*
12     *
13     */
14 }
```

Listing 7: @ManyToMany Professor

JPA ANNOTATIONS

```
1  @Entity
2  @Table(name="Course")
3  public class Course {
4    /*
5     *
6     */
7     private Collection<Professor> professors = new
       Collection<>();
8     @ManyToMany
9     @JoinTable {
10     name = "Course_Professors",
11     joinColumns = {@JoinColumn(name="professor_id")},
12     inverseJoinColumns = {@JoinColumn(name="course_id"
       )}
13    }
14    public Collection<Professor> getProfessors() {
       return professors; }
15    /*
16     *
17     */
18 }
```

Listing 8: @ManyToMany Course

JPQL - Introduction

The Java Persistence Query Language (JPQL) is a platform-independent object-oriented query language defined as part of the Java Persistence API (JPA) specification.

JPQL is used to make queries against entities stored in a relational database. It is heavily inspired by SQL, and its queries resemble SQL queries in syntax, but operate against JPA entity objects rather than directly with database tables.

JPQL - Read

Given a student id (*studId*) find the exams the student has not yet passed.

```
1 SELECT e
2 FROM Exam e
3 WHERE (SELECT count(r)
4   FROM Registration r
5   WHERE r.student.id = :studId AND
6         --studId is a parameter
7         r.exam.id = e.id OR
8         (r.exam.course = e.course AND r.grade IS NOT
    NULL)) = 0);
```

JPQL - Create

Insert a new registration: first we obtain the corresponding student and exam and then we create a new registration.

```
1  public void insertRegistration(int studentId, Exam
       examDetached, @Nullable Integer grade){
2    /***/
3    entityManager = factory.createEntityManager();
4
5    Student student = entityManager.getReference(Student
       .class, studentId);
6    Exam exam = entityManager.getReference(Exam.class,
       examDetached.getId());
7    Registration registration = new Registration(student
       , exam, grade);
8
9    entityManager.getTransaction().begin();
10   entityManager.persist(registration);
11   entityManager.getTransaction().commit();
12   /***/
13   entityManager.close()
14 }
```

JPQL - Update

Update an existing registration inserting the grade.

```
1  public void updateRegistration(Registration reg, int
       grade) {
2    reg.setGrade(grade);
3    try {
4      entityManager = factory.createEntityManager();
5      entityManager.getTransaction().begin();
6      entityManager.merge(reg);
7      entityManager.getTransaction().commit();
8    } catch (Exception ex) {
9      ex.printStackTrace();
10     System.out.println("A problem occurred inserting a
        registration!");
11     throw ex;
12   } finally {
13     entityManager.close();
14   }
15 }
```

JPQL - Delete

Delete a registration.

```
1  public void deleteRegistration(int studentId, Exam
       exam) {
2    try {
3      entityManager = factory.createEntityManager();
4      entityManager.getTransaction().begin();
5      Query query = entityManager.createQuery("DELETE
       FROM Registration r WHERE r.exam = :exam AND r.
       student.id = :studId");
6      query.setParameter("exam", exam);
7      query.setParameter("studId", studentId);
8      query.executeUpdate();
9      entityManager.getTransaction().commit();
10   } catch (Exception ex) {
11     ex.printStackTrace();
12     System.out.println("A problem occurred inserting a
        registration!");
13     throw ex;
14   } finally {
15     entityManager.close();
16   }
17 }
```