University of Pisa

SCUOLA DI INGEGNERIA
Corso di Laurea in Artificial Intelligence and Data Engineering

Hibernate tutorial

Candidati
Alice Nannini
Giacomo Mantovani
Marco Parola
Stefano Poleggi

Relatore
Prof. Pietro Ducange

Anno Accademico 2019–2020

Indice

1 Introduction

Hibernate is a middleware Java framework, which provides an Object-Relational Mapping (ORM). The term object-relational mapping refers to a programming technique used to combine software applications, based on the object-oriented programming (OOP) paradigm, with systems for managing relational databases (RDBMS). It means that it offers a model for mapping objects belong to the programming domain into entities belong to the relational model domain.

Later we will describe the development of an application using the build automation tool Maven, although it is possible to create and organize a project by hand, downloading and including .jar files manually. Moreover in this tutorial, we will use the MySql platform for the database, but you can choose any relational database (Oracle, SQL Server, PostgreSQL, etc).

2 Application setup

The first step to develop an application using Hibernate is to create a new Maven project on your favorite IDE and add the dependency to the *pom.xml* file. Surely you have to include hibernate dependency and db-connector for the database you want to use (Mysql in our case) in addition to eventual other dependencies you will use during the development of your application.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0" +
                            "http://maven.apache.org/xsd/maven-4.0.0.xsd">
         <modelVersion> 4.0.0 </modelVersion>
         <groupId> Task1 </groupId>
         <artifactId> Task1 </artifactId>
         <version> 0.0.1-SNAPSHOT </version>

         <name> Hibernate Hello World Program </name>

         <build>
                 <defaultGoal> install </defaultGoal>
                 <sourceDirectory> src </sourceDirectory>
                 <plugins>
                         <plugin>
                            <artifactId>maven-compiler-plugin</artifactId>
                            <version> 3.8.0 </version>
                            <configuration>
                                    <release> 11 </release>
                            </configuration>
                         </plugin>
                 </plugins>
         </build>

         <dependencies>

                 <dependency>
                         <groupId> org.hibernate </groupId>
                         <artifactId> hibernate-core </artifactId>
                         <version> 5.4.4.Fina l</version>
                 </dependency>

                 <dependency>
                         <groupId> mysql </groupId>
                         <artifactId> mysql-connector-java </artifactId>
                         <version> 8.0.17 </version>
                 </dependency>

         </dependencies>
</project>
```

Another fundamental step to do, during the setup phase, is to create a new database and insert all information related in various configuration files.

3 Hibernate configuration

At this point we start the configuration of the application; in order to complete this step, we have to create a standard file in the /src/META-INF folder of the project: *persistence.xml*. Its aim is to provide all the information about the saving, updating, and interrogating of the database and some information related to the mapping.
Here we define the persistence-unit; the parameters of this XML file are called properties, in which we can specify:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence" +
        "␣http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
<persistence-unit name="studentsratings">

 <properties>
  <property name="javax.persistence.jdbc.url"
        value="jdbc:mysql://localhost:3306/studentsratings?serverTimezone=UTC">
  </property>
  <property name="javax.persistence.jdbc.user" value="root" />
  <property name="javax.persistence.jdbc.password" value="" />
  <property name="javax.persistence.jdbc.driver"
        value="com.mysql.cj.jdbc.Driver" />
  <property name="hibernate.hbm2ddl.auto" value="update" />
  <property name="hibernate.show_sql" value="true" />
  <property name="hibernate.format_sql" value="true" />
  <property name="hibernate.transaction.jta.platform"
      value="org.hibernate.service.jta.platform.internal.WeblogicJtaPlatform"/>
 </properties>

</persistence-unit>

</persistence>
```

4 Basic operation

As said before, Hibernate is a middleware platform, so it is interposed between the java program and the database. It's possible to associate its behavior with that of a cache, in which an object can be inserted into a persistence context temporarily, waiting to be brought in a permanent condition into the database. So the life cycle of an object is the following:

- the object is allocated into the memory during the java program execution: at this point, it is in the 'New state' or the 'Transient state'
- the object can be promoted to the persistence state, calling the persist method of the entity manager. Please note, at this point, the object is not synchronized with the database yet
- the object is synchronized with the database during the flush time phase, using the commit method of a Transaction or the flush method of the EntityManager

Summarizing, a PersistenceContext can be denied as context, in which the objects are waiting for propagation of the changes to the database. This functionality is realized by the EntityManager. To work using this architecture is necessary to allocate an EntityManager, thanks to the EntityManagerFactory, as shown in the following code.

```
private static EntityManagerFactory factory;
private EntityManager entityManager;

public void setup() {
        factory = Persistence.createEntityManagerFactory("db_name");
}

    ...

public void exit() {
        factory.close();
}
```

5 Mapping

Hibernate takes care of the mapping from Java classes to database table and he must know in which table a class have to be mapped. This functionality can be done in two ways:

- XML mapping
- Java annotation

In this tutorial, we are going to show the mapping using the annotation approach; in order to use it, we import the javax.persistence library. You just add an @Entity annotation to the class and an @Id annotation to the primary key attribute (or before every attribute that composes the primary key). Hibernate maps the entity to a database table with the same name and uses a default mapping for each attribute. It is important to observe that after the class declaration, Hibernate will check the existence of the corresponding table and will create it if it doesn't exist. In order to manage the ID field, Hibernate give us the possibility to generate values automatically, specifying the @GeneratedValue annotation, with different strategies. We use the option "strategy = GenerationType.IDENTITY", which indicates that the persistence provider must assign primary keys for the entity using a database identity column. Other possible strategies are AUTO, SEQUENCE, and TABLE, for more details, we invite you to read the official documentation. You can also choose a different name for the table, adding the @Table(name = "table_name") annotation.

```
@Entity(name = "Degree")
@Table(name = "degree_programmes")
public class Degree {

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

        @Column(name = "name", unique = true)
        private String name;
    ...
}
```

Moreover, you can map an attribute of a class in a field of the table, using the @Column annotation and specifying some other information and options (name of the column, length, etc) and more.

6 CRUD operation

In this chapter we describe the CRUD operations (Create, Read, Update, Delete) and how we can implement them in hibernate.
All these operations are perfomed following the schema shown in the code below.

```
try {
        entityManager = factory.createEntityManager();    [1]
        entityManager.getTransaction().begin();    [2]

            < CRUD operatations >

        entityManager.getTransaction().commit();    [3]

    } catch (Exception ex) {
        ...
    } finally {
        entityManager.close();    [4]
}
```

[1] Instantiate a new persistence context (EntityManager).
[2] Start a transaction
[3] Propagate the changes to the database
[4] Close the EntityManager to release its persistence context and other resources

Please note: [2] and [3] operations must not be executed in case of READ operations since there are no changes to be made permanent on the database.

6.1 Create

To perform this operation is necessary to istantiate a new object of a class mapped on the database [1] and to make the object persistent inside the transaction [2].

```
Degree degree = new Degree(0, "degree_name");    [1]
try {
        entityManager = factory.createEntityManager();
        entityManager.getTransaction().begin();
        entityManager.persist(degree);    [2]
        entityManager.getTransaction().commit();
    }
    ...
```

6.2 Read

There are mainly two ways to compute this operation: Retrieve an object executing a query using different SQL query languages, the examples in this tutorial refer to the HQL.
This snippet of code shows the query declaration by the EntityManager createQuery function [1] and the related parameter setting [2]. The query is executed calling the getResultList function, that builds a list by resultset.

```
Student student = null;
String selectUser = "SELECT u" +
                    "FROM Users u" +
```

```
                    "WHERE username = ?1" +
                    "AND password = ?2";

try {
        entityManager = factory.createEntityManager();

        TypedQuery<Student> query = entityManager.createQuery(selectUser,
                                                    Student.class);   [1]
        query.setParameter(1, username);   [2]
        query.setParameter(2, password);

        List<Student> results = query.getResultList();
        student = results.get(0);
        }
    ...
```

The other way is to retrieve an object by a primary key using the EntityManager find function [1].

```
try {
        entityManager = factory.createEntityManager();
        Student student = entityManager.find(Student.class, StudentId); [1]
}
  ...
```

6.3 Update

To compute this task, it is necessary to retrieve an object, that will be already persistent, from the database. Once the object is retrieved, changes are made on the fields using the set methods [1]. To make the changes effective, it is necessary to use the commit method [2].

```
try {
    entityManager = factory.createEntityManager();

        < retieve an object from database professorCommet >

    entityManager.getTransaction().begin();
    professorComment.setText(text);   [1]
    professorComment.setDate(date.toString());
    entityManager.getTransaction().commit();   [2]
}
  ...
```

6.4 Delete

To compute this task, it is necessary to retrieve the object to eliminate from the database, that will be already persistent. Once the object is retrieved, deleting is made on the object using the EntityManager remove method [1]. To make the changes effective, it is necessary to use the commit method [2].

```
try {
    entityManager = factory.createEntityManager();

        < retieve an object from database professorCommet >
```

```
        entityManager.getTransaction().begin();
        entityManager.remove(professorComment);
        entityManager.getTransaction().commit();
    }
    ...
```

7 Association mapping

To model the relationship between two database tables as attributes in your domain model, Hibernate provides us association mapping features, such as:

- One-to-One
- One-to-Many
- Many-to-Many

7.1 One-to-One

The code below shows how to set a One-to-One mapping between Subject and Professor.

```
@Entity(name = "Professor")
@Table(name = "professor")
public class Professor {

    @Id
    @GeneratedValue
    private int id;
    private String name;

    @OneToOne(mappedBy = "professor", cascade = CascadeType.ALL,
              fetch = FetchType.LAZY)
    private Subject subject;

        public Professor() {};
        //Constructor, getters and setters
}

@Entity(name = "Subject")
@Table(name = "subject")
public class Subject {
        @Id
        @GeneratedValue
        private int Id;

        private String name;

        @OneToOne(fetch = FetchType.LAZY)
    @MapsId
    private Professor professor;

        public Subject() {}

        // constructor Getters and setters
}
```

Using the "mappedBy" value of the annotation @OneToOne lets hibernate know that the key for the relationship is on the other table, since only one table has a foreign key constraint to the other one. In this way hibernate can still link the table not containing the constraint to the other one. The MapsId annotation provides mapping for the primary key of the parent entity. The "CascadeType.ALL" value means that any change happened on Professor must be cascaded to Subject as well. The "FetchType.LAZY" value means that when you load up an instance of the

Subject via a Hibernate query, Hibernate will not load this professor unless you explicitly ask it to.

7.2 One-to-Many

The code below shows how to set a One-To-Many mapping between Subject and Professor, supposing that a professor can be the owner of multiple subjects, but a subject is associated with at most one professor.

```
public class Subject {
        ...

        @ManyToOne(fetch = FetchType.LAZY)
        @JoinColumn(name = "profId")
        private Professor prof;

        //constructors, getters and setters
        ...
}

public class Professor {
        ...

        // relation with profComments.
        @OneToMany(mappedBy = "prof", cascade = CascadeType.ALL,
                                        fetch = FetchType.LAZY)
        private List<Subject> subject = new ArrayList<Subject>();

        public Professor() {}

        //constructor, getters and setters
}
```

Using the @OneToMany and @ManyToOne annotations the bidirectional relation between the object is defined. If the @JoinColumn annotation is not specified, hibernate will create three tables instead of two. The third table (Professor_Subject) is used to store the foreign keys. By using the @JoinColumn association, hibernate creates a new column in the subject table named "profId" which stores a foreign key to the professor table.

7.3 Many-to-Many

The code below shows how to set a Many-to-Many mapping between Subject and Professor, supposing that a professor can be the owner of multiple subjects and a subject can be associated with multiple professors.

```
public class Subject {
        ...

        @ManyToMany(cascade = CascadeType.ALL)
        @JoinTable(name = "teaching", joinColumns = @JoinColumn(name = "subjectId")
                , inverseJoinColumns = @JoinColumn(name = "profId"))
        private Set<Professor> professor = new HashSet<Professor>();
```

```
        //constructors, getters and setters
        ...
}

public class Professor {
        ...

        // relation with Subjects.
        @ManyToMany(mappedBy = "professor", cascade = CascadeType.ALL)
        private Set<Subject> subject = new HashSet<Subject>();

        public Professor() {}

        //constructor, getters and setters
}
```

In this case, the relation between Professor and Subject is defined by the @ManyToMany annotations in both classes. This association has two sides: the owning side and the inverse side. The owning side is the Subject and the join table is defined by the owning side with the @JoinTable annotation, named "teaching" in this example. The @JoinColumn annotation is defined to link the column with the main table.