Assignment 6

Web Architectures 2018-2019

Nicolò Pomini

November 9, 2018

1 Introduction

The goal of this assignment is to create two applications that save and read some objects to/from a database, to guarantee data persistance. In particular, two types of objects are requested, with a *one to many* relationship between them.

The problem statement also says that one application deals with creating and saving data, while the other one reads data from the database. Both applications uses *Hibernate* technologies.

2 Explanation

To create a *Hibernate* application, a specific pattern should be followed in order to do so, which is now illustrated in all its steps.

Firstly, a database to save persistently data has to be created. For this kind of application, a relational database is chosen: it has a very simple structure, with two entities and a *one to many* relationship between them, as can be seen in the entity-relationship model, in Figure 1. The translation to the relational model is the following:

$$Writer(id, name, surname)$$
$$Book(id, author, title, publicationYear)$$

$$Book.author = fk(Writer.id)$$

where *Writer.id* is the primary key for the relation *Writer*, and *Book.id* is the primary key for the relation *Book*.

The next step is creating the so called *POJO* classes, that are standard Java classes that map the relations showed above.

In addition to the previous classes, an XML file – the *Hibernate mapping* file – is created. This file maps every *POJO* class with a certain relation in the database, and it also translates every *POJO* class attribute to the respective relation field.

Another XML file is needed – the *Hibernate configuration* file – to specify with which type of database the application is connected, where the database is located and which are the credentials to start a connection.

Finally, the logic of the application is developed, which deals with all the operations to save and retrieve data to/from the database.

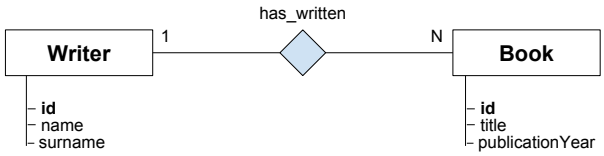

Figure 1: The ERM of the database with which the applications work.

3 Implementation

The implementation follows the order previously described in the Explanation section.

Firstly, the relational database that is chosen is `MySql`. To create the database, a SQL file – called `db.sql` – is used. It deals with creating a new database – called `webarch6` – selecting the brand new just created database and creating the two tables that translate the relational model. These tables are called `writer` and `book` respectively. In particular, is very important to give a default value to the *foreign key* in table `book` – the field `book.author` – which identifies the primary key of a writer – otherwise Hibernate will face an error.

The code is organized in three different packages, each one with a name in the form `it.unitn.disi.webarch.ass6.<name>`. In the following explanations only the `<name>` part will be considered to refer to a package. The project was implemented using `Maven` technology, to make things easier with external dependencies (in this case the Hibernate library and the JDBC connector for the database).

The *POJO* classes are developed in the `model` package. They are two java files, one for the `writer` table and one for the `book` table, called `Writer.java` and `Book.java` respectively. They have a structure which is similar to a *Java Bean*. All attributes are private, there is one attribute for each attribute in the database, but the relationship between the two objects is reversed than how it is in the database structure: in fact, the class `Book.java` has no fields about the *author* of the book, while the class `Writer.java` has a set, which represents the books written by a writer. In addiction to that, all the attributes have their own getters and setters, and there are two constructors: one with parameters – used by the implemented programs to create instances of objects – and one without parameters – used by Hibernate.

Since a `Maven` project was created, the configuration files are placed in the `src/main/resources` folder. There are two configuration files:

- the *Hibernate mapping* file – called `Writer.hbm.xml` – which maps the two *POJO* classes to their respective database table, and the attributes of the class with the respective attributes on the database. For example, the mapping between the `Writer.java` class and the `writer` table is done with the following XML entry

  ```
  <class name = "it.unitn.disi.webarch.ass6.models.Writer"
         table = "writer">
  ```

  and the map between attributes is done in this way

```
<property name = "name" column = "name" type = "string"/>
```

A particular mapping is reserved for the relation between the two table, and it is translated with the following entry:

```
<set name = "books" cascade="all">
    <key column = "author"/>
    <one-to-many class="it.unitn.disi.webarch.ass6.models.Book"/>
</set>
```

- the *Hibernate configuration* file – called `hibernate.cfg.xml` – in which is written what type of database is used, which are the login credentials to access to it, which is the URL of the database and which is the mapping file.

In the package `dbmanager` is placed the class `DBManager.java`, which provides the functions to add new writers to the database, and to retrieve some data from the same source. To do so, it uses mainly a `SessionFactory` and a `Transaction` objects – imported from `org.hibernate`. In particular, to add a writer, all its parameters must be provided – a name, a surname and a set of books. Then, a `Writer` object is created and saved to the database. The following code does this operation:

```
public Writer addWriter(String name, String surname, Set books){
    Session session = this.factory.openSession();
    Transaction tx = null;
    Writer writer = null;

    try {
        tx = session.beginTransaction();
        writer = new Writer(name, surname);
        writer.setBooks(books);
        Integer writerID = (Integer) session.save(writer);
        tx.commit();
        writer.setId(writerID);
    } catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    } finally {
        session.close();
    }
    return writer;
}
```

The useful thing of Hibernate is that it handles the insertion of both `Writer` and `Book` objects into the database.

A similar operation is done to retrieve data from the database. Also in this case, just specifying the `id` of a writer, a complete `Writer` objects with all the related `Book` objects is returned. The code is the following:

```
public Writer getWriter(int id) {
    Session session = this.factory.openSession();
    Transaction tx = null;
    Writer w = null;
    try{
        tx = session.beginTransaction();
        String query = "FROM Writer W WHERE W.id = " + id;
        List writers = session.createQuery(query).list();
        for (Iterator i = writers.iterator(); i.hasNext();) {
           w = (Writer) i.next();
        }
        tx.commit();
    } catch(HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    } finally {
        session.close();
    }
    return w;
}
```

The `DBManager.java` class offers also methods to select all the writers, and to get the books written by a given writer after a certain threshold year.

In the package `programs`, the two required programs are implemented.

In particular, the file `Maker.java` deals with creating 3 writers with random strings as name and surname, and with a random number of books related with them. For every writer, its name and surname are generated, then a set of books is created (also here with random titles and random publication years), and finally the method `addWriter` explained before is called.

The file `Reader.java` decides a threshold year – 2000 for instance – then gets all the writers, and for each writer calls the method `getWrittenAfter`, offered by the `DBManager.java` class, which gives back only the books written after the given threshold. These books are printed to the standard output.

Finally, since a `Maven` project was used, the `pom.xml` file has to be mentioned. Inside this file all the depencencies are specified – the Hibernate core library and the connector for MySql – with the following XML entries:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.3.7.Final</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>6.0.6</version>
</dependency>
```

In addition to that, a `plugin` tag was added to ensure that the jar file generated by `Maven` contains all the external dependencies.

4 Deployment

To deploy the project, several steps are needed.

First of all, it is assumed that `MySql` and `Maven` are installed on the machine.

Let us place in the assignment directory with a terminal session, and open the `MySql` prompt, with the command

```
mysql -u <username> -p
```

and insert the password. At this point the `db.sql` file is loaded, and the database and its structure are created:

```
\. db.sql
```

Now it is possible to close the `MySql` prompt – with `ctrl+d` – and always using the terminal open the `Assignment6` directory, which contains the source code and the configuration files of the project. To let `Maven` to collect the dependencies and to compile the source files, the following command is required:

```
mvn package
```

A small premise before starting with the execution: since the output contains many messages generated by Hibernate, only the output of the `Maker` and `Reader` classes is reported – simply copying and pasting it – instead of place screenshots inside this document.

Before launching the program, the tables in the database are empty, as can be seen via terminal:

```
mysql> select * from writer;
Empty set (0.00 sec)

mysql> select * from book;
Empty set (0.00 sec)
```

Now let us launch the `Maker` class, with the command

```
java -cp target/Assignment6-1.0-SNAPSHOT-jar-with-dependencies.jar
         it.unitn.disi.webarch.ass6.programs.Maker
```

which produces – among several Hibernate messages – the following output:

```
Writer added
Writer(name: fa420864, surname: 50b37da1, books:
        [Book(id: 1, title: 156abbc2, pubYear: 2005)
         Book(id: 2, title: 3ec430b2, pubYear: 1989)
         Book(id: 3, title: f24a3e83, pubYear: 1998)]
Writer added
Writer(name: 3a73d279, surname: b43d9725, books:
        [Book(id: 4, title: 4db9683c, pubYear: 1983)
         Book(id: 5, title: ee51fe7a, pubYear: 1997)]
Writer added
Writer(name: 64effeab, surname: 355b5cf4, books:
        [Book(id: 6, title: 9a09ad01, pubYear: 2008)
         Book(id: 7, title: 22ad1388, pubYear: 2003)]
```

Now it is possible to check with the `MySql` prompt that the same data is saved also in the database:

```
mysql> select * from writer;
+----+----------+----------+
| id | name     | surname  |
+----+----------+----------+
|  1 | fa420864 | 50b37da1 |
|  2 | 3a73d279 | b43d9725 |
|  3 | 64effeab | 355b5cf4 |
+----+----------+----------+
3 rows in set (0.00 sec)

mysql> select * from book;
+----+----------+--------+-----------------+
| id | title    | author | publicationYear |
+----+----------+--------+-----------------+
|  1 | 156abbc2 |      1 |            2005 |
|  2 | 3ec430b2 |      1 |            1989 |
|  3 | f24a3e83 |      1 |            1998 |
|  4 | 4db9683c |      2 |            1983 |
|  5 | ee51fe7a |      2 |            1997 |
|  6 | 9a09ad01 |      3 |            2008 |
|  7 | 22ad1388 |      3 |            2003 |
+----+----------+--------+-----------------+
7 rows in set (0.00 sec)
```

Finally, let us launch the `Reader` class with the command

```
java -cp target/Assignment6-1.0-SNAPSHOT-jar-with-dependencies.jar
         it.unitn.disi.webarch.ass6.programs.Reader
```

which displays all the books written after the year 2000 by every writer. The following is the output:

```
List of books written after 2000 by writer with id 1:
Book(id: 1, title: 156abbc2, pubYear: 2005)

List of books written after 2000 by writer with id 2:

List of books written after 2000 by writer with id 3:
Book(id: 6, title: 9a09ad01, pubYear: 2008)
Book(id: 7, title: 22ad1388, pubYear: 2003)
```