Assignment 7

Web Architectures 2018-2019

Nicolò Pomini

November 17, 2018

1 Introduction

The goal of this assignment is to create an enterprise application that models an university scenario. In particular, a server application is asked, which let the user insert new students, new professors, new courses and new exams, to print inserted data, to enroll students in courses and exams, to print lists of enrolled students in a certain course or exam, and to add marks to an exam.

Furthermore, the assignment statement asks to implement the server using EJB technology, and the client with a standard Java application.

2 Explanation

First of all, an assumption to simplify the scenario was made during the design of the assignment. The statement says that a student can *take* and being *enrolled* to a corse, without saying nothing that differentiate these two statuses. For semplicity, no difference between the two states was modelled, so when a student takes a course means also that she is enrolled in it.

To implement this assignment, several steps are needed. The modeled scenario is quite complex, because there are several entities that interact with each other.

Firstly, a database – to save persistently data – has to be created. A relational database is chosen, with five entities:

- a generic *Person*, which models a common base for both students and professors;
- a *Professor*, which has no further attributes than a generic *Person*, but it is related with a single *Course*;
- a *Student*, which adds an attribute – called *matricola* – to the standard *Person* structure. A student can take n courses and n exams;
- a *Course*, which is taught by a *Professor* and has a numerical ID and a name. A course is taken by n students, and is related with n exams;
- an *Exam*, which is related to one *Course* and is performed by n students. It has a date and a numerical ID. The relation with *Student* has an additional attribute, called *mark*.

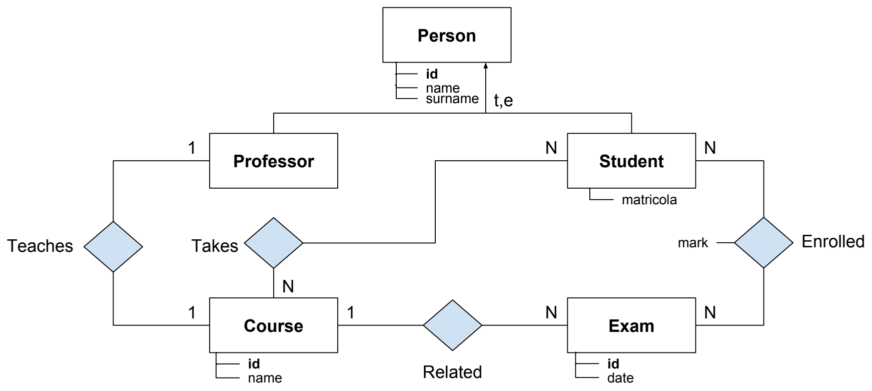

Figure 1: The ERM of the database with which the applications work.

The whole entity-relationship model can be seen in Figure 1. The translation to the relational model is the following:

$$
\begin{aligned}
&Professor(\underline{id}, name, surname)\\
&Student(\underline{id}, name, surname, matricola)\\
&Course(\underline{id}, name, professor)\\
&Takes\_course(\underline{student, course})\\
&Exam(\underline{id}, date, course)\\
&Enrolled(\underline{student, exam}, mark)
\end{aligned}
$$

with the following constraints:

$$
\begin{aligned}
&Unique(Student.matricola)\\
&Course.professor = fk(Professor.id)\\
&Takes\_course.student = fk(Student.id)\\
&Takes\_course.course = fk(Course.id)\\
&Exam.course = fk(Course.id)\\
&Enrolled.student = fk(Student.id)\\
&Enrolled.exam = fk(Exam.id)
\end{aligned}
$$

where also the underlined attributes are the primary keys for their respective relations.

Since the problem statement is not asking anything that implies that each person must be uniquely identified, the relation *Person* in the ERM does not appear in the relational model above, but it is embedded in both *Student* and *Professor* relations. In ohter words, it is not necessary that a professor ID is unique also among student IDs.

The assignment is implemented in two different projects: one for the client, one for the server. Due to the problem statement, the implementation of the assignemt is such that the server depends from the client. The reason is simple:

since the beans of the server are invoked remotely from the client, the interfaces of these beans are defined into the client. So the server beans need the client interfaces to be implemented.

Let us focus on the server. The code is organized in three main sections: one that takes care of the structure of the data modelled in this scenario, defining types and methods for each existing object; one that deals with the core functions, to read and write data; one that provides the business methods, or the functions made available by the application.

The relational model written above is translated into a set of Java classes on the server side. These classes represent the entities modelled into the assignment, and they deal with mapping either attributes and relations with the other classes with the physical model of the database. To do so, Hibernate annotations are used. In this way the interface between the server and the database is completely handled by the Hibernate library, that creates the corresponding queries to read and write data from the database. In this way, when an ojbect is requested from the database, not only the object itself is returned, but also all its relations with the other objects. For example, when a *Student* is asked to be retrieved, a complete Student object is returned, containing also the list of courses and exams she is enrolled in.

The logic for reading and writing data to/from the database is implemented in a set of *DAO* classes, that are standard Java classes in which the operations for creating and saving new data – and also for reading existing data – are written.

The bean classes expose methods that are called by the client, so they define the business logic of the application. For each bean, a remote interface is placed into the client – which contains the signatures of all the available methods for that bean – and a Java class that implements the bean is placed inside the server. The methods exposed by the beans call the functions made available by the *DAO* classes.

Finally, the client is a simple Java application that uses *JNDI* to retrieve the available beans, and consequently uses the remote methods to allow the user to insert and read data. It provides a simple command line interface through standard input / standard output, to let the user invoke the remote methods of the server.

3 Implementation

In this section the details of the previous section are provided.

First of all, 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. The name of the tables correspond to the name used in the relational model above.

3.1 Server

Let us now focus on the server. It is implemented in a Java Enterprise Application, using `Maven` to simplify dependencies with external libraries. The application – called `Assignment7Server` – contains two modules:

- the `ejb` module, which contains all the source code and the configuration files for Hibernate;
- the `ear` module, that depends from the `ejb` module and contains the `.ear` file, which is used to deploy the whole server to an application server.

Since the `ear` module does not contain source code, this section is focused on the `ejb` part.

The code is organized in packages, each one in the form `it.unitn.disi.-webarch.ass7.server.<package-name>`. From now on only the `<package-name>` will be mentioned.

The entities classes are placed into the `entities` package. Each entity class has as attributes all the attributes of the respective table, plus all the relations that the database table has with other tables. The latter are translated using *Collections* or *Sets* of other entity objects, together with Hibernate annotations. An annotation is a syntacticall metadata added above the definition of these kind of attributes, that are used by Hibernate to understand how the database is structured, and consequently to build queries. For example, the `Course` class has a *one to many* relationship with `Exam` objects, a *many to many* relationship with `Student` objects and a *one to one* relationship with `Professor`. These relationships are annotated in the following way:

```
@OneToMany(fetch = FetchType.LAZY, mappedBy = "course")
private Collection<Exam> exams;

@ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
@JoinTable(name = "takes_course", joinColumns = {
    @JoinColumn(name = "course", nullable = false)
}, inverseJoinColumns={@JoinColumn(name="student",nullable=false)})
private Collection<Student> students;

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

The first one says that the attribute `Exam.course` maps the relationship, the second one indicates that there is a table in the database – called `takes_course` – used to implement this relationship, and the third one says that the field `professor` is used to map the relationship with a `Professor` object.

Another very important annotation concerns the primary key of each table. It is translated into every entity class with the tag

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

In general, every entity class has several things in common:

- the annotated relationships with the other class;
- the standard attributes, annotated with `@Column(name="attr_name");`

- an empty constructor, plus an optional constructor with parameters – to be used in other classes of the server;
- public getters and setters for each attribute;
- the class implements `java.io.Serializable`.

To make Hibernate work in the correct way, the project contains also the `hibernate.cfg.xml` file, placed in `src/main/resources` folder of the `ejb` module. It contains the information about which type of database is used – in this case is `MySql` – but also the keys to access to the database and the fully qualified name of the mapping class – which are those described above.

Let us now focus on the logic of the application. The function to read and write data on the database are implemented in the `dao` package. For each resource – `Student`, `Professor`, `Course` and `Exam` – a Java class called `<resource-name>DAO` is created. In these classes all the methods to fulfill the operation requested by the assignment statement are placed. For example, in the class `CourseDAO`, the fuctions for adding a new course, getting all existing courses, enrolling a new student to the course and getting the list of enrolled students are implemented. For example, to enroll a student to the course the following function is implemented:

```
public boolean enrollStudentinCourse(int courseId, int studentId) {
    Session session = this.factory.openSession();
    Transaction tx = null;
    boolean result = false;
    try{
        tx = session.beginTransaction();
        Course course = (Course) session.get(Course.class, courseId);
        Student student = (Student) session.get(Student.class, studentId);
        course.getStudents().add(student);
        session.update(course);
        tx.commit();
        result = true;
    }catch(HibernateException | NullPointerException e) {
        if (tx!=null) tx.rollback();
    } finally {
        session.close();
    }
    return result;
}
```

which opens an Hibernate session to create new data to be saved. In particular, a `Student` and a `Course` are requested to Hibernate, and then the student is added to the list of student of the course. Finally, the transiction with the database is committed and the data is saved back to the database. In general all the other DAO function are very similar to this one, so there is an opening of a connection, which is used to get data or to open a transaction for saving data.

The business logic is exposed in the bean classes, which are in the `beans` package. Each bean is stateless, because the operations available in this application does not request to maintain a status with the user. Furthermore, every bean implements a *remote class*, which is placed in the client project.

Even in this case there is a bean for each resource. The exposed functions are very simple, and they use the functions made available by the DAO class. For example, the function used to get all the students enrolled in a certain course is implemented in the `CourseBean` class, in the following way:

```
@Override
public String getEnrolledStudents(int courseId) {
    CourseDAO dao = new CourseDAO();
    Collection<Student> s = dao.getEnrolledStudents(courseId);
    if(s == null || s.isEmpty())
        return "No students are enrolled in course " + courseId;
    else {
        String rtr = "Students enrolled into course " + courseId + ":\n";
        for(Student c: s) {
            rtr += c.toString() + "\n";
        }
        return rtr;
    }
}
```

which uses the `getEnrolledStudents` function of `CourseDAO` and returns a string containing all the enrolled students in the given course.

Finally, since the server is created using a `Maven` project, the file `pom.xml` – with all the dependencies – has to be mentioned. It contains dependencies with the client application, the Hibernate core library, the Java EE api and the MySql connector.

3.2 Client

Let us now focus on the client. It is a Java application – also created using `Maven` – organized in two packages:

- one that contains the remote interfaces of the server beans, called `it.unitn.disi.webarch.ass7.client.interfaces`. The interfaces are used by the client to know which methods the server makes available, and they are used to let the user to perform the actions requested by the problem statement;
- the other that contains the *real* client, called `it.unitn.disi.webarch.ass7.client`. It contains two classes – `BaseAccess`, which is a sort of utility class for retrieving server beans via JNDI, and `Client`, which is the entry point of the application. The latter prints a list of possible operations the user can perform, and uses the remote bean function with the input provided via the standard input to fulfill the user requests.

Finally, since the client app is also a `Maven` project, it has a `pom.xml` file with two dependencies: one for the Java EE API, and the other for the Wildfly client library.

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 `Assignment7` directory. Since the server depends from the client, the latter has to be compile before the former. The following operation has to be done:

```
cd Assignment7Client
mvn package
cd ..
```

which creates a Jar file for the Client, and in the end brings the current location back to the assignment directory.

The server can now be compiled, starting from the `ebj` module:

```
cd Assignment7Server/Assignment7Server-ejb
mvn package
cd ..
```

which also creates a Jar file for the `ejb` module and in the end brings the current location back to the assignment directory.

Now it is time to compile also the `ear` module, moving into its folder and calling again maven:

```
cd Assignment7Server-ear
mvn package
```

At this point is possible to deploy the server into Wildfly. Let `SERVER_HOME` be the server project home directory (the `Assignment7Server` in this case) and `WILDFLY_HOME` the home directory of the server application. To make the deployment is necessary to copy the `.ear` file of the server into the deployment folder of Wildfly, doing the following command:

```
cp SERVER_HOME/Assignment7Server-ear/target/
                Assignment7Server-ear-1.0-SNAPSHOT.ear
   WILDFLY_HOME/standalone/deployments
```

At this point Wildfly produces some outputs in its console, indicating also the available beans.

Since the client was compiled before, it can be launched with the command

```
cd Assigment7Client
java -cp target/Assignment7Client-1.0-SNAPSHOT-jar-with-dependencies.jar
                it.unitn.disi.webarch.ass7.client.Client
```

At this point a list of options that can be performed is printed out. This list associates a letter from a to n (plus the q letter to quit) to an action required by the problem statement, so for example the command a is to insert a new student, b to insert a new professor, and so on until n which prints marks assigned to an exam. To select an operation is necessary to write the corrisponding letter and press enter. To make the report more compact and more understandable, the output of some operations are now pasted here.

First of all, some professors are added, with command b:

```
Please select an action to perform
[List of actions...]
b
Insert professor's name:
Marco
Insert professor's surname:
Ronchetti
Inserted: Professor(id: 1, name: Marco, surname: Ronchetti)
```

After some iterations, all professors are printed, with command f:

```
Please select an action to perform
[List of actions...]
f
All professors:
Professor(id: 1, name: Marco, surname: Ronchetti)
Professor(id: 2, name: Roberto, surname: Sebastiani)
Professor(id: 3, name: Alberto, surname: Montresor)
Professor(id: 4, name: Renato, surname: Lo Cigno)
```

In a similar way, a couple of students, some courses and some exams are inserted. This is the situation:

```
All students:
Student(id: 1, name: Nicolo', surname: Pomini, matricola: 203319)
Student(id: 2, name: Pinco, surname: Pallino, matricola: 123456)

All courses:
Course(id: 1,name: Programmazione 1)
Course(id: 2,name: Algoritmi e strutture dati)

All exams:
Exam(id: 1, date: 2018-12-20)
```

Now is possible, for example, enroll the two students to the exam, to assign their marks and to print all the marks of the exam, with commands k, m and n respectively. For instance, let us assign a mark $= 25$ to student with $ID = 2$: firstly, the student is enrolled to the exam; secondly, the mark is assigned; finally, all marks are printed – in this case only one student is enrolled to the exam, so only one mark is printed.

```
k
Insert exam ID:
```

```
1
Insert student ID:
2
Student 2 enrolled in exam 1

m
Insert exam ID:
1
Insert student ID:
2
Insert mark:
25
Mark assigned

n
Insert exam ID:
1
Marks for exam 1:
Mark for student2: 25
```

A similar thing can be done with courses: it is possible to enroll student to a course and to print the list of student enrolled to a course, with commands *i* and *j* respectively.

5 Comments and notes

During the making of this assignment, several issues were faced.

First of all, many problems with Hibernate annotations were encountered: all the unidirectional relationship explained in the course slides did not work for me. For example, the *one to one* relation between *Professor* and *Course* was always leaving the foreign key of the course table set to `NULL`, while the *one to many* between *Course* and *Exam* was crashing the server. These issues were solved using bidirectional translations.

Secondly, when a remote bean was created – and consequently a remote interface was added into the client project – the `pom.xml` of the client project was automatically filled with a version of Java EE API that was different from the version into the server: in fact, on the client was set the Java EE 6 API, while on the server Java EE 7 API. Of course this difference made the execution of the project fail, until the misalignment was found.

Thirdly, issues with JNDI binding were faced, precisely with the path to specify in the `context.lookup` function. In fact, the course slides[1] says to use the string in the form `java:jboss/exported...`, while the Jboss documentation[2] says to use the string in the form `"ejb:" + appName + ...`. None of the two worked for me, I used a string without prefixes, in the form `appName + "/" + moduleName + ...`, that made everything work fine.

Last but not least, the `mvn package` command in the client project was not working. In fact, during the creation of the jar file with dependencies, the pro-

[1]week 6, EJB – Introduction to EJB – Slides-1, page 32

[2]docs.jboss.org/author/display/WFLY10/EJB+invocations+from+a+remote+client+using+JNDI

cess failed due to a stack overflow. A piece of code of an external library (`org.-codehaus.plexus.archiver.AbstractArchiver$1.hasNext` for instance) was looping endlessly. This problem was solved making clear – into the `pom.xml` file of the client – the `maven-assembly-plugin` version, setted to 2.6, and increasing the stack size to 10 megabyte, using the tags

```
<jvmArgs>
    <jvmArg>-Xms10m</jvmArg>
    <jvmArg>-Xmx10m</jvmArg>
</jvmArgs>
```