Assignment 8
Web Architectures 2018-2019

Nicolò Pomini

December 26, 2018

1 Introduction

The goal of this assignment is to create a web application that models the same scenario of the assignment 7. It is an university scenario, in which is possible to 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.

In particular, the assignment statement asks for a `Spring I/O` based web application, with the GUI made as a web interface, and not like a command line interface.

2 Explanation

Since the scenario is the same of the previous assignment, the structure of the database on which the web server relies is the same of the previous one. To distinguish them, its name is changed, becoming *webarch8* (instead of *webarch7*). The same assumptions made about the modeling phase persist from the previous report. For the sake of clarity, the entity-relationship model and the relational model are reported in this report. The former can be seen in Figure 1, while the latter is the following:

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

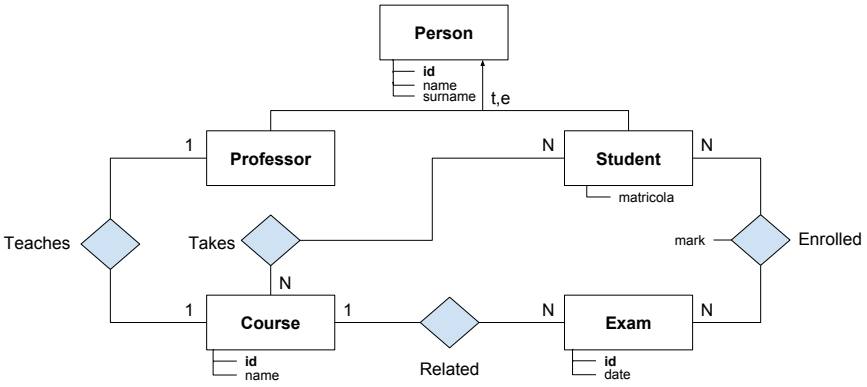

Figure 1: The ERM of the database with which the applications work.

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.

The assignment is implemented in two different projects: one for the client, one for the server. It is thought to be potentially deployed in three different locations, since every component is as indipendent as possible from each other. The only *agreements* that are needed are the following:

- the database must be a relational one – `MySql` for instance – and the server must know its URL and the credentials to log into it;
- the server provides some API, which are used by the client.

In this way, each component can be physically placed in different locations, and they comunicate only via HTTP protocol.

Let us focus on the server: it is a `Spring` based application, leaning on a relational database and offering a set of RESTful API. The code is organized in four 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 offers a set of Java interfaces – one for each relevant resource – that are translated by `Spring` into some beans; one that provides the exception classes, to handle error related to not found data; one that defines the API. The latter is the endpoint of the application, or what the client interfaces with.

The API offered by the server are organized by resources – `Student`, `Professor`, `Course` and `Exam`. They are not *pure* RESTful, to avoid an endless recursion during the display of these data. For example, a student has a list of courses that she is enrolled in, and a couse has a list of enrolled students. If the API were RESTful at all, there would be a list of courses for each student, which would contains a list of student, each one with its list of courses and so on, causing an endless loop. So, some resources leave out some attributes.

For reasons of compactness, the API specifications are at the bottom of the report, in Section 5. Since the problem statement is the same of the previous assignemnt, there is no need to identify which user performs the operations: for this reason the parameters of the API concern only the university resources, and they can be used by anyone.

Finally, the client is a simple web application, made only of HTML, CSS and Javascript. In this way, it can be exectuted on any device with a web browser, without any particular dependency or configuration. Furthermore, it is responsive, so it can be used with the maximum comfort on a small device, such as a mobile phone. For this feature, a very famous and widely used library is chosen, called `Bootstrap`[1].

3 Implementation

The assignment is implemented in two separate projects: one for the client, one for the server. In this section some impementation details are provided: firstly about the server, and secondly about the client.

3.1 Server

The server project is a standard Java project. This because Spring allows to create a web application as if it was a standard application, with a `main` method that deals with starting a web server and deploying the project on it. A maven project was used, in order to make the managements of the required libraries easier.

The server relies on a relational database, which 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 tables that translate the relational model. The name of the tables correspond to the name used in the relational model above.

The code is organized in packages, each one in the form `it.unitn.disi.webarch.ass8.server.<package-name>`. The main class is placed into the *standard* package (without any `<package-name>`), in the file called Application.java. From now on only the `<package-name>` will be mentioned to refer to a specific package.

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. They are the same of the previous assignment: standard Java classes with Hibernate annotations. They are placed in the package `entities`.

[1] getbootstrap.com

In the package `repositories` are placed a set of Java interfaces – one for each type of Hibernate object used by the application – that extend the class `CrudRepository`. This class wants two generic types: the objects on which it operates, and the key of these objects. For example, the interface related to the Student class extends the `CrudRepository<Student, Integer>` class, because it works with Student objects, which have an integer key on the database. This set of interfaces is translated into a set of beans at runtime by Spring, and each of them provides the CRUD operations – create, read, update, delete – on its objects.

In the package `exceptions` are placed a set of classes used by the API in case some operations go wrong. In particular, each exception class is a standard Java class that extends RuntimeException. There is also a class called `Advice`, that in case one of the exception is thrown, it makes sure that the API either print the corrisponding error message and send the right HTTP error code. For example, to bind a `StudentNotFoundException` with a 404 error, the following code is necessary:

```
@ResponseBody
@ExceptionHandler(StudentNotFoundException.class)
@ResponseStatus(HttpStatus.NOT_FOUND)
String studentNotFoundHandler(StudentNotFoundException ex) {
        return ex.getMessage();
}
```

Finally, the API are implemented into the package `controllers`, which contains one controller class for each resource provided by the web service. A controller maps a specific base URL, and it is defined with the following annotations:

```
@Controller
@RequestMapping(path="/student")
public class StudentController { ... }
```

In this way, the class `StudentController` maps the `/student` address.

On the mapped address, several services are available, each one identified by the pair <URL, HTTP method>, according to the API specifications. For example, the class `StudentController` has three endpoints:

1. </student, POST>, for adding a new student;
2. </student, GET>, to get all the students;
3. </student/{studentid}, GET>, to get the student with id `studentid`.

Each endpoint is implmented in a similar way: some are very easy, some other are more complex because they deals with more than one resource at the same time. In general, they make use of the Spring annotations. For example, the following code implements the endpoint number 3 of `StudentController`:

```
@GetMapping(path="/{id}")
public @ResponseBody Student  getStudent(@PathVariable Integer id) {
    return repository.findById(id).orElseThrow(
            () -> new StudentNotFoundException(id)
    );
}
```

where the annotation indicates that this method is used on the specified path. It uses the repository related to the Students, to search for a student with the given id. In case a student with that id does not exist, a `StudentNotFoundException` is thrown.

A more complex API is the one to enroll a student to a course. It is implemented in this way:

```
@PostMapping(path="/{courseid}/enroll/{studentid}")
public @ResponseBody Course enrollStudent(@PathVariable("courseid") int courseid,
                                          @PathVariable("studentid") int studentid) {
    Student s = studRepository.findById(studentid).orElseThrow(
            () -> new StudentNotFoundException(studentid)
    );
    Course c = repository.findById(courseid).orElseThrow(
            () -> new CourseNotFoundException(courseid)
    );
    Collection<Course> studentCourses = s.getCourses();
    studentCourses.add(c);
    s.setCourses(studentCourses);

    Collection<Student> courseStudents = c.getStudents();
    courseStudents.add(s);
    c.setStudents(courseStudents);

    studRepository.save(s);
    return repository.save(c);
}
```

It is invocated when a POST request to the specified path comes. At the beginning, it search for the corresponding student and course, and in case at least one of the two does not exist an exception is thrown. After that, the course is added to the list of courses taken by the student, and viceversa. Finally, both the student and the course are updated, with the method `save` of the related repository class.

As a very last thing, the assignment is designed so that the client and the server can be placed in two different servers, with different domains and different ports. This means that the server has to accept Cross-Origin requests (CORS). For this reason, an additional package – called `filters` – is created. It contains a class that acts as filter, adding the header `Access-Control-Allow-Origin` to each response. Furthermore, every API implemented in the `controllers` package has an additional annotation, `@CrossOrigin`, that allows cross origin requests.

3.2 Client

The client project contains simple HTML and Javascript files, which implement five different web pages: one that acts as the landing page, and it contains four button that forwards to the respective other four, and the other pages offer each one the functionalities related to a resource. In particular, there is a page for the students, one for the professors, one for the courses and one for the exams. Every page displays the existing data, and gives to the user the possibility to

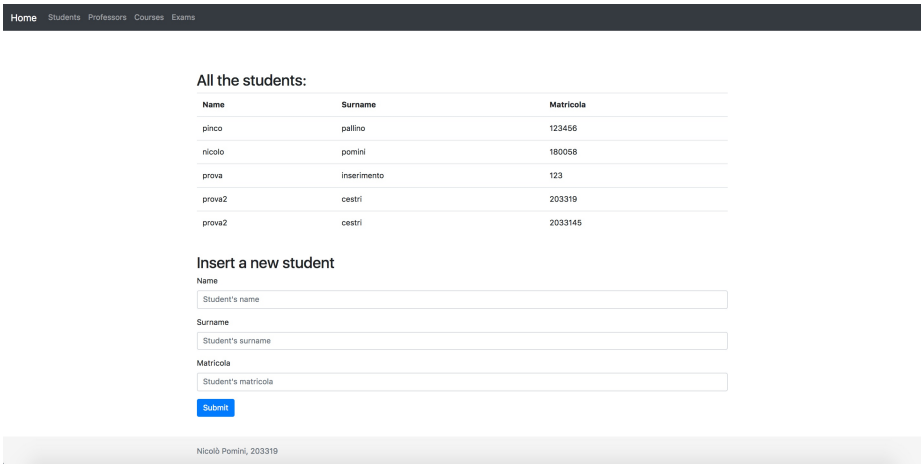

Figure 2: The layout of the student page.

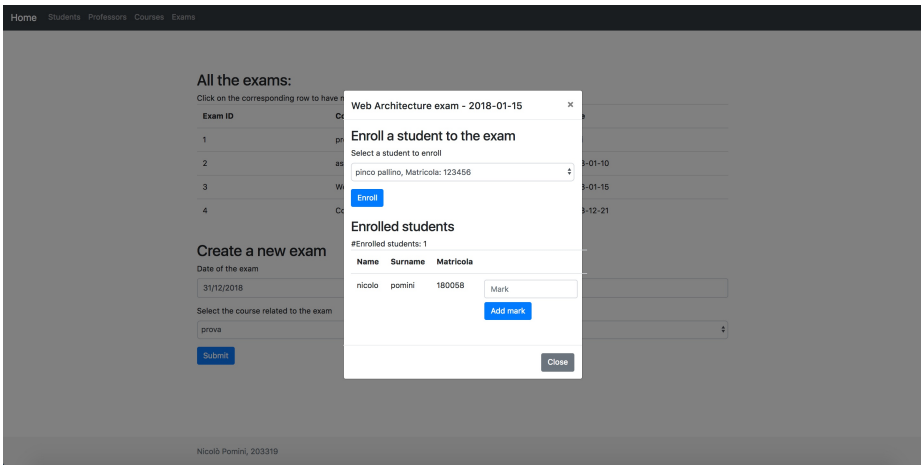

Figure 3: The layout of the exam page.

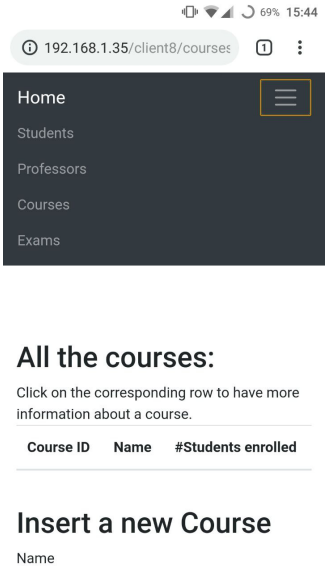

Figure 4: The layout on a mobile phone.

insert some new data into the system. Furthermore, the course and exam pages have more complex functionality, such as enrolling a student to a specific course or exam, or showing the list of enrolled student to a particular course/exam, or adding a mark to a student enrolled to an exam.

Since the assignment does not require to identify an user, or to grant some functionality only to certain users, the client does not use cookies, and the interface is the same for every user. So everyone can add courses or exam, or enroll people to courses or exam, or add marks.

As said, every page has the basic functions of display existing data and insert new data. The display part is implemented through a table, while for inserting new data there is a specific form. An example is the student page, which can be seen in Figure 2. The course and exam pages use some more complex HTML tools – called `modals` – which are blocks of content initially hidden, and when an event occurs – like a click – they are shown in the foreground. For example, in the exam page – see Figure 3 –, a click on one row of the table that displays the data opens a `modal` that lists the students enrolled to that exam. It also offers the possibility to add a mark to an enrolled student, or to enroll a new student to the exam.

Finally, the client is responsive: this means that the layout is arranged on the screen size, offering a comfortable navigation also on mobile devices. An example of the mobile layout can be seen in Figure 4.

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. It is time to compile the server project and to launch it. To do so, open the Assignment8Server directory, call the maven script that generates the jar file and launch the web server. The maven command may take a while to be executed, since the produces jar is indipendent from any possible web server install on the machine. The following commands are needed:

```
cd Assignment8Server
mvn package
mvn spring-boot:run
```

Now the server is running, and it is possible to use the client.

For security reasons, web browsers do not allow cross origin requests, so if the client was used as a set of normal files, it would not work correctly. This because launching a HTTP request on a file exectuted as a normal file – and not run by a web server – does not produce all the required headers to overcome cross origin issues. In particular, the header Origin is needed by the server to accept the request. So, in order to use properly the client, it has to be deployed in some web server, such as Apache.

To do so, copy the whole client directory into the Apache documents directory, and then launch the server with the command

```
sudo apachectl start
```

At this point it is just necessary to open a browser, and go to
http://localhost/client8
to use the application.

5 API specifications

The API are the following:

- **Student**
  - Add a new student:
    - POST /student
    - with parameters:
      - name: the name of the student, a string
      - surname: the surname of the student, a string
      - matricola: the matricola of the student, a string
    - result:

```
{
    "id": 1,
    "name": "Nicolo",
    "surname": "Pomini",
    "matricola": "123456"
}
```

  - Get a student:
    - `GET /student/{studentId}`
    - result:

```
{
    "id": 1,
    "name": "Nicolo",
    "surname": "Pomini",
    "matricola": "123456"
}
```

  - Get all the students:
    - `GET /student`
    - result:

```
[
    {
        "id": 1,
        "name": "Nicolo",
        "surname": "Pomini",
        "matricola": "123456"
    }
]
```

- **Professor**
  - Add a new professor:
    - `POST /professor`
    - with parameters:
      - name: the name of the professor, a string
      - surname: the surname of the professor, a string
    - result:

```
{
    "id": 1,
    "name": "Marco",
    "surname": "Ronchetti"
}
```

  - Get a professor:
    - `GET /professor/{professorId}`
    - result:

```
{
    "id": 1,
    "name": "Marco",
    "surname": "Ronchetti"
}
```

- Get all the professors:
  - `GET /professor`
  - result:

```
[
    {
        "id": 1,
        "name": "Marco",
        "surname": "Ronchetti"
    }
]
```

- **Course**
  - Add a new course:
    - `POST /course`
    - with parameters:
      - name: the name of the course, a string
      - professor: the id of the professor who does the course, an integer
    - result:

```
{
    "id": 1,
    "name": "Web Architecture",
    "students": []
}
```

  - Get a course:
    - `GET /course/{courseId}`
    - result:

```
{
    "id": 1,
    "name": "Web Architecture",
    "students": []
}
```

  - Get all the courses:
    - `GET /course`
    - result:

```
[
    {
        "id": 1,
        "name": "Web Architecture",
        "students": []
    }
]
```

  - Enroll a student to a course:
    - `POST /course/{courseId}/enroll/{studentId}`, to enroll the student with `studentId` to the couse with `courseId`

    * result:
      ```
      {
          "id": 1,
          "name": "Web Architecture",
          "students": [
              {
                  "id": 1,
                  "name": "nicolo",
                  "surname": "pomini",
                  "matricola": "123456"
              }
          ]
      }
      ```
  - Get the students enrolled to a course:
    * `GET /course/{courseId}/enrolled`
    * result:
      ```
      [
          {
              "id": 1,
              "name": "Nicolo",
              "surname": "Pomini",
              "matricola": "123456"
          }
      ]
      ```

- **Exam**
  - Add a new exam:
    * `POST /exam`
    * with parameters:
      · courseid: the id of the course with which the exam is related, an integer
      · date: the date of the exam, a string
    * result:
      ```
      {
          "id": 1,
          "date": "2018-01-15",
          "course": {
              "id": 1,
              "name": "Web Architecture",
              "students": [
                  {
                      "id": 1,
                      "name": "nicolo",
                      "surname": "pomini",
                      "matricola": "123456"
                  }
              ]
      ```

```
        }
    }
```

- Get an exam:
  - `GET /exam/{examId}`
  - result:

```
    {
        "id": 1,
        "date": "2018-01-15",
        "course": {
            "id": 1,
            "name": "Web Architecture",
            "students": [
                {
                    "id": 1,
                    "name": "nicolo",
                    "surname": "pomini",
                    "matricola": "123456"
                }
            ]
        }
    }
```

- Get all the exams:
  - `GET /exam`
  - result:

```
    [
        {
            "id": 1,
            "date": "2018-01-15",
            "course": {
                "id": 1,
                "name": "Web Architecture",
                "students": [
                    {
                        "id": 1,
                        "name": "nicolo",
                        "surname": "pomini",
                        "matricola": "123456"
                    }
                ]
            }
        }
    ]
```

- Enroll a student to an exam:
  - `POST /exam/{examId}/enroll/{studentId}`, to enroll the student with `studentId` to the exam with `examId`
  - result:

```
    {
```

```
        "mark": null,
        "student": {
            "id": 1,
            "name": "nicolo",
            "surname": "pomini",
            "matricola": "123456"
        },
        "exam": {
            "id": 1,
            "date": "2018-01-15",
            "course": {
                "id": 1,
                "name": "Web Architecture",
                "students": [
                    {
                        "id": 1,
                        "name": "nicolo",
                        "surname": "pomini",
                        "matricola": "123456"
                    }
                ]
            }
        }
    }
```


- Add a mark to an exam:
  - `POST /exam/{examId}/enrolled/{studentId}`
  - with parameter:
    - mark: the mark, a string
  - result:


```
    {
        "mark": 30,
        "student": {
            "id": 1,
            "name": "nicolo",
            "surname": "pomini",
            "matricola": "123456"
        },
        "exam": {
            "id": 1,
            "date": "2018-01-15",
            "course": {
                "id": 1,
                "name": "Web Architecture",
                "students": [
                    {
                        "id": 1,
                        "name": "nicolo",
                        "surname": "pomini",
                        "matricola": "123456"
```

```
                    }
                ]
            }
        }
    }
```


- Get the students enrolled to an exam:
  - `GET /exam/{examId}/enrolled`
  - result:


```
[
    {
        "mark": 30,
        "student": {
            "id": 1,
            "name": "nicolo",
            "surname": "pomini",
            "matricola": "123456"
        },
        "exam": {
            "id": 1,
            "date": "2018-01-15",
            "course": {
                "id": 1,
                "name": "Web Architecture",
                "students": [
                    {
                        "id": 1,
                        "name": "nicolo",
                        "surname": "pomini",
                        "matricola": "123456"
                    }
                ]
            }
        }
    }
]
```


- Get a single enrollment to an exam:
  - `GET /exam/{examId}/enrolled/{studentId}`
  - result:


```
{
    "mark": 30,
    "student": {
        "id": 1,
        "name": "nicolo",
        "surname": "pomini",
        "matricola": "123456"
    },
    "exam": {
        "id": 1,
```

```
        "date": "2018-01-15",
        "course": {
            "id": 1,
            "name": "Web Architecture",
            "students": [
                {
                    "id": 1,
                    "name": "nicolo",
                    "surname": "pomini",
                    "matricola": "123456"
                }
            ]
        }
    }
}
```