Department of Information Engineering and
Computer Science
University of Trento, Italy

Web Architectures - A.Y. 2022/2023

Assignment 4

Authors:
De Menego Luca

Dec 3, 2022

Contents

1 Introduction

The fourth assignment involves the development of a web application backed by Enterprise Java Beans, where data persistency is provided by an H2 database. Hence, two projects should be delivered:

1. an EJB application served by WildFly that interacts with the H2 database and, exploiting Hybernate, exposes remote facade session beans. The exchanged objects should be DTOs (data transfer objects).
2. a web application running on a Tomcat server external to Wildfly, that uses business delegates and a specially crafted service locator to access the beans exposed by WildFly.

Two web pages should be served by the web application:

- a `Student Page` that, given a matriculation number, provides the student's anagraphic data and the list of all the courses he is enrolled in, with the corresponding marks.
- an `Advisor Choice Page` that, given a matriculation number, provides the student's anagraphic data and the names of the teachers assigned to the courses the student is enrolled in. The actual advisor choice does not need to be implemented: what matters is just showing the data.

The database should have the following entities:

- `STUDENT`: `matriculation`, `name`, `surname`;
- `COURSE`: `name`;
- `TEACHER`: `name`, `surname`.

Moreover, the following relationships should be implemented:

- a `1:1` relation between `TEACHER` and `COURSE`;
- an `N:M` relation between `STUDENT` and `COURSE`.

2 Solution

The assignment development can be seen as a three-steps process:

1. create the database tables;
2. implement the backend EJB application;
3. implement the web application served by Tomcat.

The following subsections explain each step separately.

2.1 Database set up

Following the requirements, three basic tables have been created: `STUDENT`, `COURSE` and `TEACHER`. Based on the future use, the `1:1` relation between `COURSE` and `TEACHER` was chosen to be unidirectional: starting from a course it is possible to get the corresponding teacher, but not vice versa.

Since the `N:M` relation between `STUDENT` and `COURSE` needs to also have an additional column for the vote, a join table `STUDENT_COURSE` has been created, having:

- an `N:1` relation with `STUDENT`;
- an `M:1` relation with `COURSE`;
- a composite key (`STUDENT_MATRICULATION`, `COURSE_ID`);
- an additional column `VOTE`.

In this way, the tables `STUDENT` and `COURSE` will only need to have a `1:N` relation with the `STUDENT_COURSE` table to be able to effectively get all the needed information.
The SQL commands used to generate all the tables are the following:

```
create table STUDENT (
    MATRICULATION INTEGER(32)   not null,
    NAME          VARCHAR(30)   not null,
    SURNAME       VARCHAR(30)   not null,
    primary key (MATRICULATION)
);

create table COURSE (
    ID         INTEGER      not null generated always as identity,
    NAME       VARCHAR(50)  not null,
    TEACHER_ID INTEGER,
    primary key (ID),
    foreign key (TEACHER_ID) references TEACHER
);

create table TEACHER (
    ID      INTEGER     not null generated always as identity,
    NAME    VARCHAR(30) not null,
    SURNAME VARCHAR(30) not null,
    primary key (ID)
);

create table STUDENT_COURSE (
    STUDENT_MATRICULATION INTEGER not null,
    COURSE_ID             INTEGER not null,
    VOTE                  INTEGER,
    primary key (STUDENT_MATRICULATION, COURSE_ID),
    foreign key (COURSE_ID)             references COURSE,
    foreign key (STUDENT_MATRICULATION) references STUDENT
);
```

A visual representation of all the entities involved can be seen in the diagram in Figure 1.

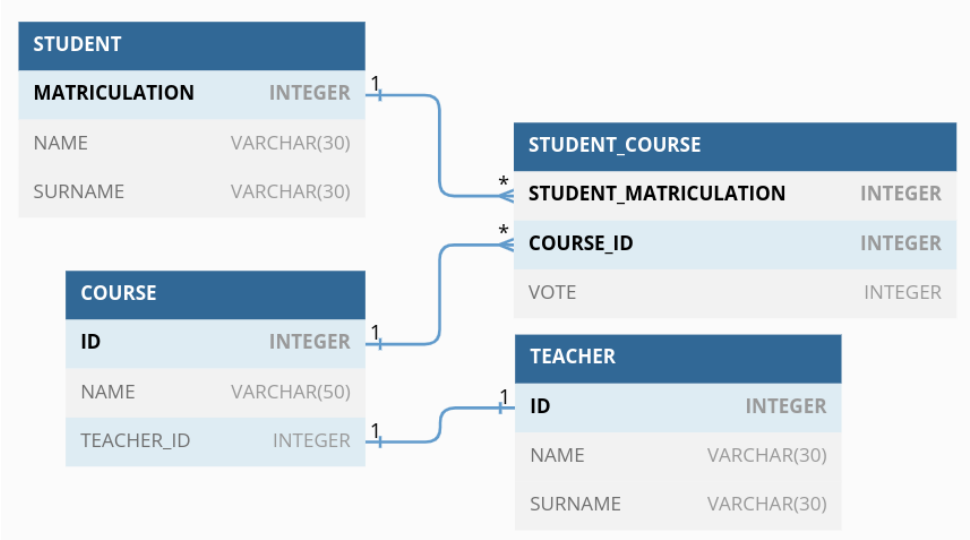

Figure 1: Diagram of the database tables and their relationships

2.2 EJB application

2.2.1 Project configuration

The EJB application has been created as a *Jakarta EE project* (Jakarta v9), with *WildFly 27.0.0 Final* as application server. Hibernate libraries have been added as dependencies, together with the `jboss-client` module. Regarding the database connection, the driver has been taken from the H2 installation folder; to make the integration work, `OLD_INFORMATION_SCHEMA=true` has been appended to the connection string, and in the advanced settings *'enable introspect using JDBC metadata'* has been checked. At this point, to allow WildFly to find and connect to the database, a new datasource has been added into the WildFly `standalone.xml` configuration file with the following properties:

```
<datasource jndi-name="java:jboss/datasources/StudsSearchDS" pool-name="StudsSearchDS"
        enabled="true" use-java-context="true" statistics-enabled="true">
    <connection-url>jdbc:h2:tcp://localhost/~/studs-search;DB_CLOSE_DELAY=-1;
        DB_CLOSE_ON_EXIT=FALSE</connection-url>
    <driver>h2</driver>
    <security>
        <user-name>sa</user-name>
        <password>sa</password>
    </security>
</datasource>
```

Eventually, in the project file `META_INF/persistence.xml` the newly created data source has been defined with:

```
<jta-data-source>java:jboss/datasources/StudsSearchDS</jta-data-source>
```

and the property `hibernate.jpa.compliance.query` has been set to false, in order to allow implicit select clauses in the project:

```
<properties>
    <property name="hibernate.jpa.compliance.query" value="false"/>
</properties>
```

As a last configuration step, a new artifact has been added to the build phase of the project: its `jar` file. This is necessary to ease the integration of the project definitions within the web application project. More information about its use can be found in Subsection 2.3.

2.2.2 Entities definition

The entities represent a way to access, update or create database records with an abstracted interface based on an object-relational mapping. In the project, each table has a corresponding entity:

- `STUDENT` -> `StudentEntity`
- `COURSE` -> `CourseEntity`
- `TEACHER` -> `TeacherEntity`
- `STUDENT_COURSE` -> `StudentCourse`

Each entity `implements Serializable`, to make it possible for it to be passed by value. As explained later, in fact, in the developed architecture these entities are passed between local and remote beans.

Moreover, each relation is defined via specially-crafted annotations (e.g. `@OneToMany(...)`) that also allow to choose various properties like `cascade` (how operations should be propagated to the other entities) or `fetch` (whether the information associated to this relation should be fetched immediately or in a lazy way). In this project, we don't want entities to be, for instance, deleted on cascade, so the `CascadeType` is always set to `CascadeType.PERSIST`. The fetch type, on the other hand, is almost always set to `FetchType.LAZY`: additional information is queried only when necessary.

As a last comment on the entities, in order to represent the composite key of the `STUDENT_COURSE` table, an `@Embeddable` class `StudentCourseId` has been created that contains two properties: the matriculation ID and the course ID. It is used in the `StudentCourse` entity as an `@EmbeddedId`. To be able to access the student and the course from an instance of this class, two properties have been defined:

```
@ManyToOne(cascade = {CascadeType.PERSIST}, fetch = FetchType.EAGER)
@MapsId("matriculation")
private StudentEntity student;

@ManyToOne(cascade = {CascadeType.PERSIST}, fetch = FetchType.EAGER)
@MapsId("id")
private CourseEntity course;
```

that are automatically mapped to the correct entities via their primary key and, being the only useful information the class exposes, automatically fetched.

2.2.3 DTOs

Various DTOs have been created, one for each type of message sent by facade session beans. In detail, we have:

- `(Student/Course/Teacher)Information`: DTOs exposing basic information about an entity (e.g. name and surname);
- `StudentAndCourses`: DTO exposing basic information about a student and all the courses he is enrolled in;

- `CourseAndTeacher`: DTO exposing basic information about a course and its assigned teacher;
- `StudentAdvisorChoices`: DTO exposing basic information about a student and all his possible advisors;
- `StudentCourseInformation`: DTO exposing basic information about a student, a course and the assigned vote;
- `EnrollmentInformation`: DTO exposing a matriculation number and a course ID, used to represent a mapping student-course;
- `CourseAndVote`: intermediate DTO, used within the `StudentAndCourses` DTO, exposing basic information about a course and the vote of a certain student.

Each DTO has a corresponding assembler that implements the following interface:

```
public interface Assembler<A, B> {
    B assemble(A entity);
}
```

Each implementation of the `assemble` method takes as input an entity and returns a DTO. Thanks to this abstraction, the session beans will only need to call a function, and the correct DTO will be created. Moreover, since most of the relations between tables are set as `FetchType.LAZY`, the assemble procedure is not only automated, but also optimized: additional information is fetched only when necessary.

Apart from the entity information, there are some DTOs that expose an additional function mapping a certain object to basic HTML code. For instance, the `StudentAndCourses` DTO has a function `mapCoursesToHTML` that creates a standard `<ul>...</ul>` list out of the courses, to limit the amount of business logic and Java code the `JSPs` will need.

2.2.4 Session beans

In this project two types of session beans have been used: `Local Stateless` and `Remote Stateless` ones. There is one local bean for each database base entity: `StudentBean`, `CourseBean` and `TeacherBean`. This beans expose basic functions to manage these entities, such as: `get<Entity>`, `add<Entity>`, `enrollToCourse`, `setVote`, `setTeacher`, etc. Each function simply returns an entity: there are still no manually crafted DTOs involved.

The other beans, the remote ones, represent the Facade beans of the project, and they are the following:

- `StudentInfoServicesBean`:
  - `getStudents()`: get all the students with basic information about them;
  - `getStudent(id)`: get a student's information and all the courses he is enrolled in;
  - `getStudentAdvisorChoices(id)`: get a student's information and all the teachers assigned to the courses he is enrolled in.
- `StudentManagementBean`:
  - `addNewStudent(student)`: create a new student;
  - `enrollToCourse(enrollmentInfo)`: enroll a student to a certain course;
  - `setVote(studId, courseId, vote)`: set a vote to a given student for a given course;
- `CourseManagementBean`:

  - `addNewCourse(course)`: create a new course;
  - `setTeacher(courseId, teacherId)`: assign a teacher to a certain course;

- `TeacherManagementBean`:
  - `addTeacher(teacher)`: create a new teacher.

They use the local beans via dependency injection with:

```
import jakarta.inject.Inject;

@Inject
Student studentBean;
```

and, after calling their functions, they use the DTOs assemblers to automatically generate the response object the client wants. For instance, let's analyze the `StudentInfoServices` two last functions: `getStudent` and `getStudentAdvisorChoices`. In the implementation, they actually use the same function of the same local bean: the function `getStudent(id)`. The only difference is the assembler and the information that is actually lazily fetched:

```
@Override
public StudentAndCourses getStudent(Integer matriculation) {
    return new StudentAndCoursesAssembler().assemble(
        studentBean.getStudent(matriculation)
    );
}

@Override
public StudentAdvisorChoices getStudentAdvisorChoices(Integer matriculation) {
    return new StudentAdvisorChoicesAssembler().assemble(
        studentBean.getStudent(matriculation)
    );
}
```

2.2.5 Fill the database

After having developed the facade session beans, a Java program `FillDatabase.java` has been written that automatically fills the database using the beans. In detail, it gets the beans via JNDI and it:

- creates 5 students;
- creates 4 teachers;
- creates 4 courses;
- assigns each course to a certain teacher;
- enrolls some students to some courses;
- assigns some votes to some students.

2.3 Web application

2.3.1 Project configuration

The web application has been again created as a *Jakarta EE project* (Jakarta v9), but in this case with *Tomcat 10* as application server. There are two main dependencies this project needs, apart from the standard ones:

- the usual `jboss-client` module;
- the EJB application described in Subsection 2.2.

As stated before, during the build phase thanks to a specially-crafted configuration the `EJB project` also outputs a `JAR` file. The most straightforward way to be able to access its class definitions is to directly include the JAR file as a project dependency. However, this is not enough: even if the IDE has now access to these definitions, the Tomcat server does not yet. To complete the configuration, the same JAR file and the `jboss-client` must be put within `WEB-INF/lib/`. In order to automate this set up in *IntelliJ*, these steps can be followed:

1. open the project settings;
2. select each `war` artifact;
3. create a class `lib` in the `WEB-INF` folder;
4. via *right-click* add a copy of these two JARs inside the folder.

Now the dependencies will be automatically injected into the `lib` folder on each build.

2.3.2 Service Locator

The developed session beans must be accessed via a lookup, providing their JNDI name and a context with the properties `INITIAL_CONTEXT_FACTORY` and `PROVIDER_URL`. Performing the lookup each time we need to access a session bean is a waste of time and resources: the developed service locator maintains a reference to the beans within a specially-crafted cache that maps a JNDI name to an `Object`. In this way, only the first request for a certain bean will fire an actual lookup.

The service locator starts with a `ejbPrefix` already set to:

```
ejb:/studs-search-backend-1.0-SNAPSHOT
```

Moreover, it provides a special `String getBeanName(Class<?> c)` function that, given a certain bean class, automatically constructs the JNDI string[1]. Thanks to this abstraction, requesting a specific session bean to the service locator simply requires a function call like the following:

```
<X> facade = (<X>) ServiceLocator.getBean(<XBean>.class);
```

For the actual lookup, the service locator uses a specially-crafted `StudsContext` class that automatically constructs the correct context in its constructor and exposes a `lookup` function of the form:

```
public Object lookup(String name) throws NamingException {
    InitialContext context = new InitialContext(properties);
    return context.lookup(commonPrefix + "/" + name);
}
```

[1]The function has been implemented based on the documentation provided at https://docs.jboss.org/author/display/WFLY10/EJB%20invocations%20from%20a%20remote%20client%20using%20JNDI.html

2.3.3 Business Delegates

Business delegate is a Java EE pattern that reduces the coupling between business services and the presentation layer, while hiding implementation details. A business delegate can be seen as an adaptor to invoke business objects from a client, and in general there should be one business delegate for each facade session bean.

Hence, in the project architecture there are four business delegates:

- `StudentInfoServicesBD`;
- `StudentManagementBD`;
- `CourseManagementBD`;
- `TeacherManagementBD`.

Each business delegate gets access to the session beans it needs in its constructor by calling the service locator, and it implements the interface implemented by the session bean itself. The exposed functions, however, will simply call the corresponding functions present in the facade bean: the client must not contain any business logic.

At this point, using the services provided by a certain session bean `<X>` will only require the client code to instantiate the corresponding business delegate.

2.3.4 Servlets and JSPs

As a good practice to follow when developing `JSPs`, all JSP pages have been put within the `WEB-INF` folder, to keep them private and only accessible through *Java Servlets*. In this way, before presenting the view to the user the Servlet can check if the request contains everything as expected.

Hence, there are three servlets, one for each developed web page:

- `HomeServlet`: a servlet that simply forwards the request to the `index.jsp`, where the user is presented with a form to search for a certain student by matriculation number.
- `StudentServlet`: a servlet that, given a student matriculation number, gets the student's anagraphic data, together with information on the courses the he is enrolled in (name and vote, if any). If no errors are found, the user is presented with the corresponding JSP; otherwise, the user is redirected to the home page with an error message.
- `AdvisorChoiceServlet`: a servlet that, given a student matriculation number, gets the student's anagraphic data, together with the list of teachers assigned to the courses the student is enrolled in. If no errors are found, the user is presented with the corresponding JSP; otherwise, the user is redirected to the home page with an error message.

After a DTO has been generated by the business delegate called by the servlet, it is inserted into the request as an attribute, so that it can later be retrieved by the JSP.

Since the returned DTOs are all simple Java Beans that implement `Serializable` and that have an empty public constructor, they can be integrated within the JSPs by simply using the `<jsp:useBean/>` tag. As an example, the JSP linked to the `StudentServlet` uses:

```
<jsp:useBean id="studentAndCourses"
    class="...studssearch.backend.ejb.dtos.StudentAndCourses"
    scope="request"/>
```

Following this example, information about the student can be found by calling the corresponding getter function:

```
<%= studentAndCourses.getName() =>
```

and the list of courses, as already described in Subsection 2.2.3, can be automatically generated with:

```
<%= studentAndCourses.mapCoursesToHTML() =>
```

The user interface has been first designed with Figma, following the neumorphism trend. The design file can be visualized here. The UI has then been translated into CSS and integrated within the web application.

3 The application running

The home page, visible in Figure 2 shows a form in which the user can search for a certain student by matriculation number. If the provided value is not a number, the user will be presented with an error message (Figure 3). The same happens if the user with the given matriculation number is not found (Figure 4).

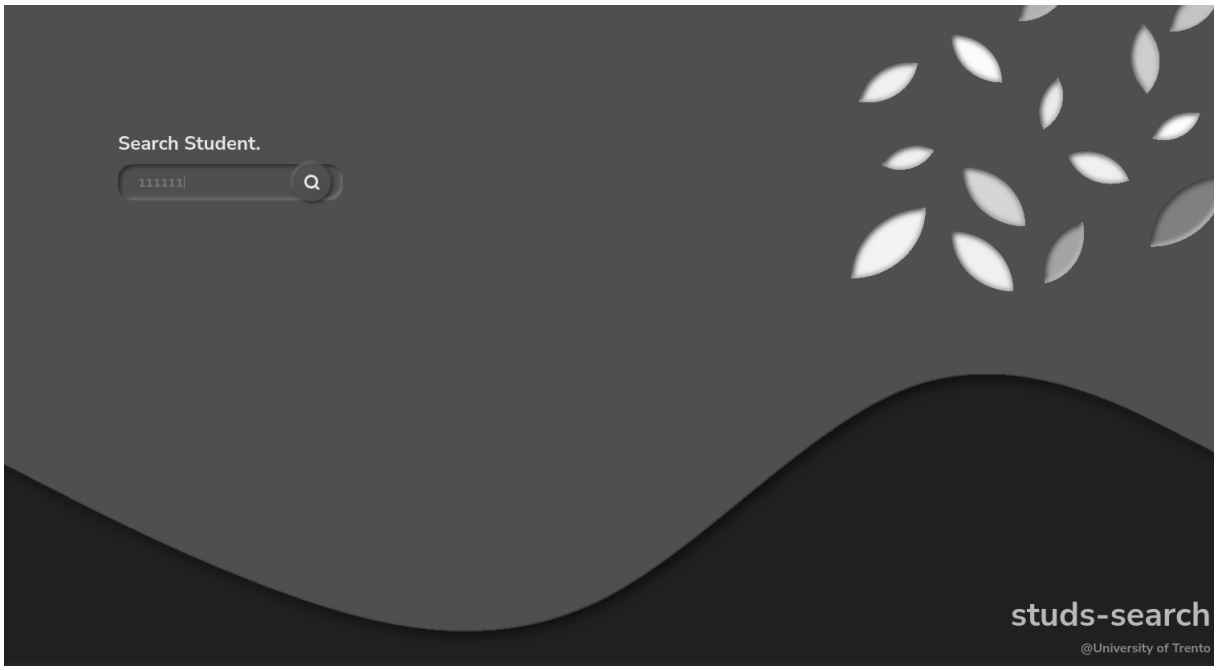

Figure 2: The home page of the web application

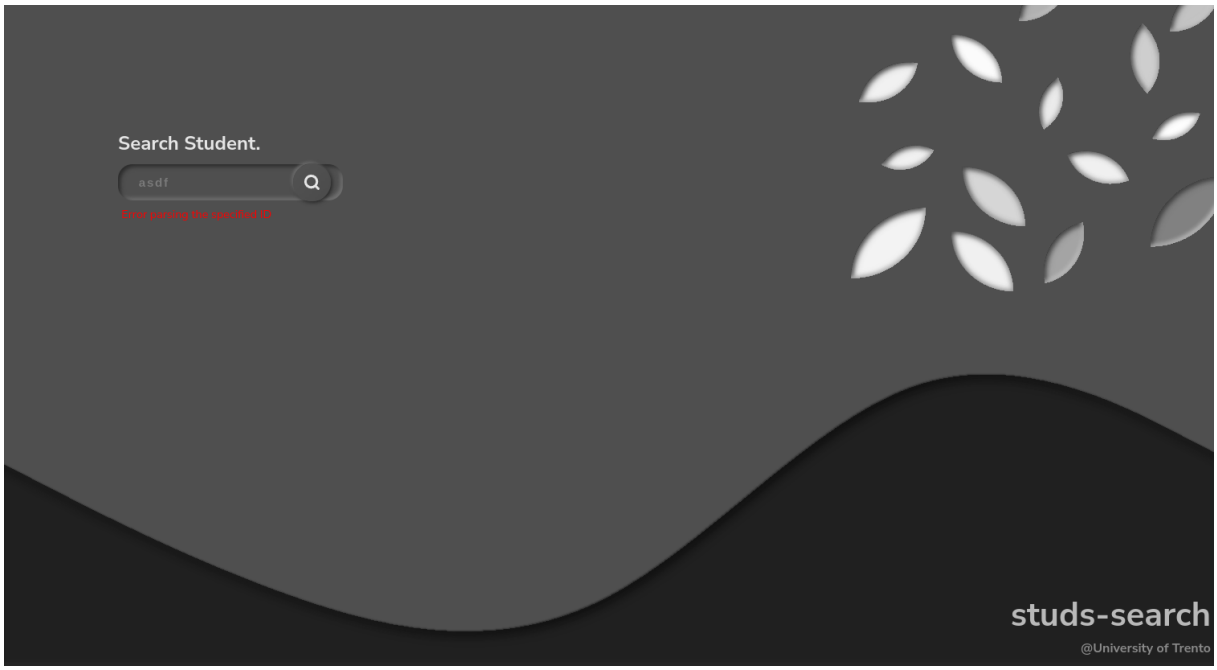

Figure 3: Error shown when the input is not a valid number

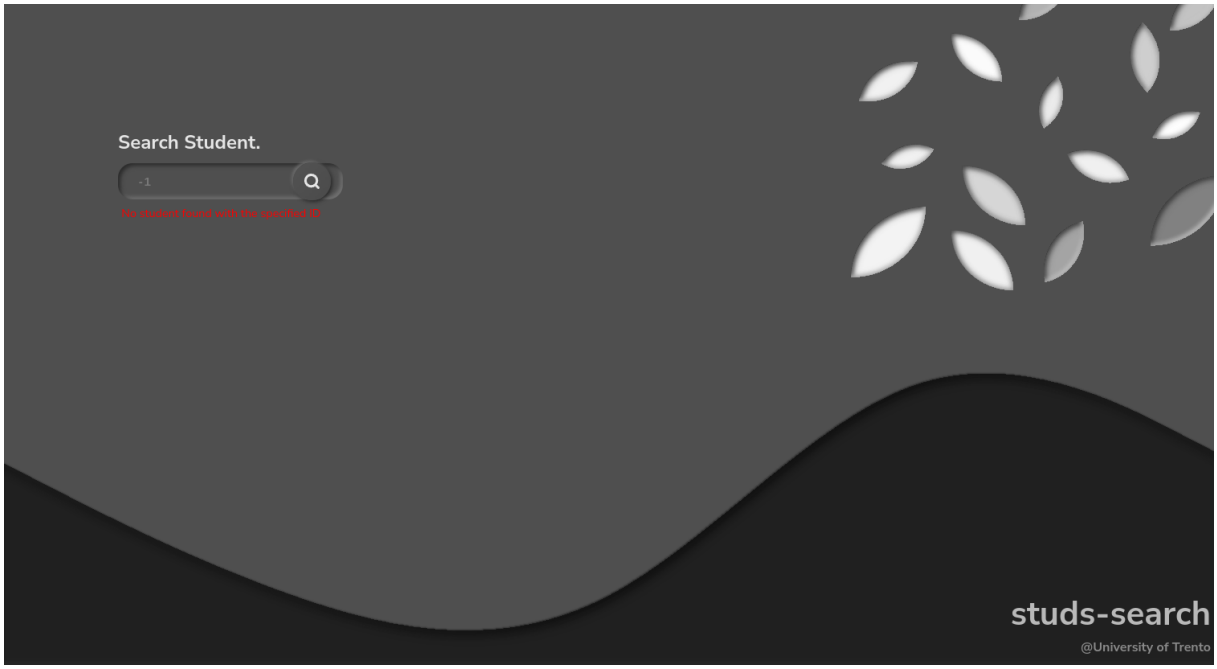

Figure 4: Error shown when no student was found with the specified matriculation number

Figure 5 shows the main student page, where anagraphic data of the user is presented, together with all the courses the student is enrolled in and the vote, if available. The button *"Choose student advisor"* allows the user to move to the next page.

Figure 5: Page showcasing the student's anagraphic data and the courses he is enrolled in, together with the respective votes

Figure 6 shows the last developed page that provides the same anagraphic data about the user and the list of teachers assigned to the courses the student is enrolled in. The actual choice of the advisor is not implemented, as it was not needed for the scope of the assignment.

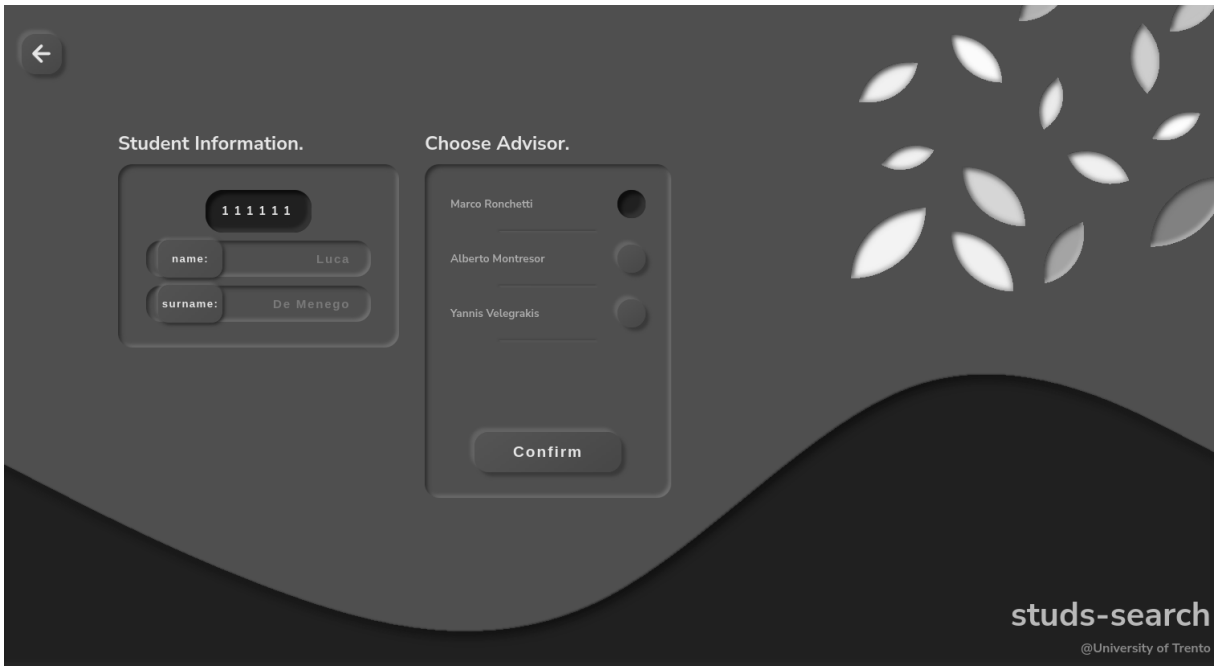

Figure 6: Page showcasing the student's anagraphic data and the teachers assigned to the courses the student is enrolled in

4 Comments

Most of the facade session beans (and the corresponding business delegates) were not in the requirements; in fact, only the `StudentInfoServicesBean` is actually used by the web application. The choice of implementing these additional beans was mainly due to the fact that I wanted to test every aspect of the backend with a simple Java program. Having all the `ManagementBeans` allowed me to easily check that all the relations were correctly implemented while also automatically filling the database with sample data (Subsection 2.2.5).