

Web Architectures - Delivery 4 v2

Marrocco Simone

February 6, 2023



Contents

0	Differences to the previous version	2
1	The assignment	3
2	Database	3
3	Architecture	3
4	Entities	5
4.1	The Student Entity	5
4.2	The Teacher Entity	6
4.3	The Course Entity	6
4.4	The StudentCourse Entity	7
5	EJBs	8
5.1	StudentEJB	8

5.2	StudentCourseEJB	9
5.3	TeachersEJB	10
6	Patterns	11
6.1	ServiceLocator	11
6.2	FacadeEJB	12
7	Front End	13
7.1	Business Delegate	13
7.2	Html Pages	16
7.3	Pages Images	19
8	Appendix A: Database Initialization	21

0 Differences to the previous version

In the previous version it was not clear the role of the Business Delegate, which led to confusing parts mix between backend and frontend, like html being served from the server or html generating functions in the business delegate class (which was wrongly named Facade, leading to more confusion). After discussing with the professor, the code is now much more understandable and the various roles better divided. It was removed from the backend the HTML parts, and in the frontend it was correctly added a Business Delegate which just exposes the backend methods.

For this document, the difference is in the Front End chapter, which substitutes the previous chapters "Html Presentation" and "Client Tomcat" and where the new Front End part is explained. The architecture image was also updated. We hope the new version is clearer.

1 The assignment

The scope of this assignment is creating a web application backed by Enterprise Java Beans (EJBs) that interacts with a database via Java Entities to show a web page shown by a separated server (in a distributed fashion).

2 Database

The database has three entities (Students, Courses and Teachers) and two relationships (Student-Courses is N:M and Courses-Teachers is 1:1). Here is how the schema was created.

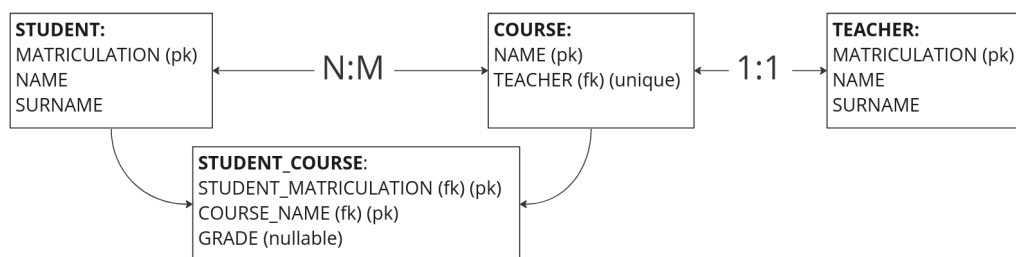


Figure 2.1: The database schema

While Teachers and Courses have a 1:1 relationship and so could be represented in a single table, it was decided to divide them for better maintainability in case in the future it is wanted to update this relationship to a 1:N one. The uniqueness of the relationship is saved in the foreign key of Course.

It was also created a join table between Students and Courses in order to save their relationship and its Grade column.

In particular, the choice of using Course.Name as primary key will help us later in the queries we need to execute, even if may not be the optimal choice in case we have multiple courses with the same name (we will suppose we won't, for now).

At the end of this document, the queries used to initialize the database will be provided.

3 Architecture

Our architecture will try to be as modular as possible, so to be better scalable and maintainable. We will have three main servers: the one that serves the client; the one that handles the business logic; and the H2 database

server. The database was linked to the Wildfly server via modifications in the standalone.xml file. The Clients connects to the Server using JNDI.

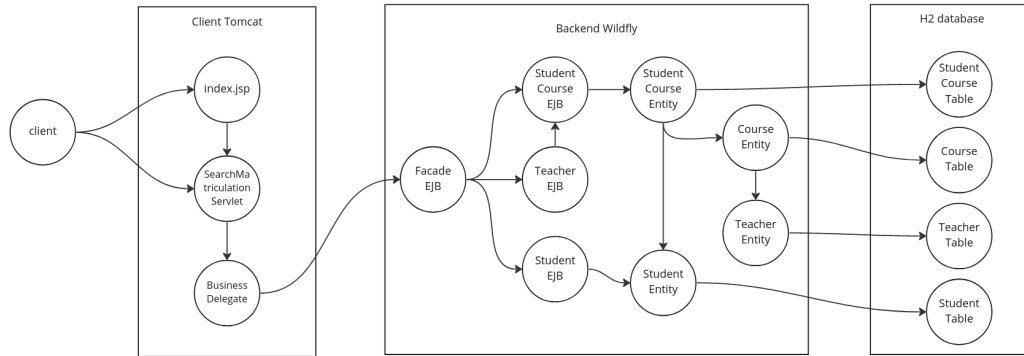


Figure 3.1: The distributed architecture

In the image, we can see how every bean and entity interacts with each other. We will explain later how each one does.

4 Entities

Entities are our link between the business logic and the database. They allow us to write much simpler queries and get data in a Java-like fashion. We will now look how they are interconnected.

For each entity, here we will omit the standard constructors, the getters/setters and the equals/hashCode, but they are present in the full code inside the project. These functions are auto generated from our IDE IntelliJ.

4.1 The Student Entity

```
1  @Entity
2  @Table(name = "STUDENT")
3  public class StudentEntity implements Serializable {
4      @Id
5      @Column(name = "MATRICULATION", nullable = false)
6      @GeneratedValue(strategy = GenerationType.IDENTITY)
7      private Integer matriculation;
8      @Column(name = "NAME", nullable = false)
9      private String name;
10     @Column(name = "SURNAME", nullable = false)
11     private String surname;
12 }
13
```

Code 1: The Student Entity

This entity (and the next one) are very basics, since they are just a translation of their schema into Java code. We can see how we are telling the Dependency Injector that all the fields are not nullable (like we did in the database schema).

4.2 The Teacher Entity

```
1  @Entity
2  @Table(name = "TEACHER")
3  public class TeacherEntity implements Serializable {
4      @Id
5      @Column(name = "MATRICULATION", nullable = false)
6      @GeneratedValue(strategy = GenerationType.IDENTITY)
7      private Integer matriculation;
8      @Column(name = "NAME", nullable = false)
9      private String name;
10     @Column(name = "SURNAME", nullable = false)
11     private String surname;
12 }
13
```

Code 2: The Teacher Entity

4.3 The Course Entity

```
1  @Entity
2  @Table(name = "COURSE")
3  public class CourseEntity implements Serializable {
4      @Id
5      @Column(name = "NAME", nullable = false)
6      private String name;
7
8      @JoinColumn(name = "TEACHER", nullable = false)
9      @OneToOne(cascade = {CascadeType.REMOVE})
10     private TeacherEntity teacher;
11 }
12
```

Code 3: The Course Entity

Here we can see how we wrote the foreign key column Teacher as a Join-Column: what this does is allowing the code to automatically join the tables and give us the already complete object it references (in this case the Teacher-Entity and not a simple Integer) instead of having us manually search for it (similar to Mongoose.populate for MongoDB in Javascript). This will be really useful for later, since we won't need many WHERE clauses in the queries. We also add the Cascade property to eliminate the Course record if the corresponding Teacher is eliminated (although we won't operate such functions).

4.4 The StudentCourse Entity

```
1  @Entity
2  @Table(name = "STUDENT_COURSE")
3  public class StudentCourseEntity implements Serializable
4  {
5      @Id
6      @JoinColumn(name = "STUDENT_MATRICULATION", nullable =
7      false)
8      @ManyToOne
9      private StudentEntity student;
10     @Id
11     @JoinColumn(name = "COURSE_NAME", nullable = false)
12     @ManyToOne
13     private CourseEntity course;
14     @Column(name = "GRADE", nullable = true)
15     private Integer grade;
16 }
```

Code 4: The StudentCourse Entity

Here we do the same thing with the columns StudentMatriculation and CourseName, only this time with a ManyToOne relationship instead of a OneToOne one. This is because we could have many time the same student and/or the same course. We can also see how the Grade column is nullable.

5 EJBs

For our EJBs, we need first to think about our logic: what we want to achieve and which queries we need. We need to create two pages: each one asks us the anagraphical data of a student, given its matriculation number, and then either the list of its courses and their grades or the list of its professors. We can see how to general request is simply a search on the Student table based on the matriculation number, while the specific ones need their specific queries.

On the naming convention: we call the interface by a simple name (like Students) and their bean implementation as the same name plus "EJB" (like StudentsEJB). In the code below we won't show the interface, since it is easily guessable (they are made of the public EJB methods).

5.1 StudentEJB

For the general one, we can create a Student EJB that does exactly that: a simple query that returns our searched Student. If we receive the error NoResultException, it means the query was unsuccessful: we return null and make the caller deal with that.

```
1  @Stateless
2  @Local(Students.class)
3  public class StudentsEJB implements Students{
4      @PersistenceContext(unitName="default")
5      private EntityManager entityManager;
6
7      @Override
8      public StudentEntity getSingleStudent(int
matriculation){
9          try {
10             Query q = entityManager.createQuery("From
StudentEntity where matriculation = " + matriculation);
11             StudentEntity s = (StudentEntity) (q.
getSingleResult());
12             return s;
13         } catch (NoResultException e) {
14             System.out.println("Student was not found");
15             return null;
16         }
17     }
18 }
19
```

Code 5: StudentsEJB

5.2 StudentCourseEJB

For the search of a student's courses, normally we would need to join the three tables Student, Course and StudentCourse. However, we can do something better: the student matriculation number and the name of the course are both fields already present in our StudentCourse table. So the query is as easy as the one before.

```
1  @Stateless
2  @Local(StudentCourse.class)
3  public class StudentCourseEJB implements StudentCourse {
4      @PersistenceContext(unitName="default")
5      private EntityManager entityManager;
6
7      @Override
8      public List<StudentCourseEntity> getStudentCourses(int
matriculation) {
9          try {
10             Query q = entityManager.createQuery(
11                 "From StudentCourseEntity where student.
matriculation = " + matriculation
12             );
13             List<StudentCourseEntity> sc = q.getResultList();
14             if (sc.isEmpty()) System.out.println("
StudentCourse is empty");
15             return sc;
16         } catch (NoResultException e) {
17             System.out.println("StudentCourse was not found");
18             return null;
19         }
20     }
21 }
22
```

Code 6: StudentCourseEJB

If instead we wanted the class Course to have a numerical ID and use it as a foreign key inside StudentCourse, the query would still remain the same: as we said before, StudentCourseEntity already has as one of its fields not the simple foreign key but the entire reference object, as we will show with the next query.

5.3 TeachersEJB

```
1  @Stateless
2  @Local(Teachers.class)
3  public class TeachersEJB implements Teachers {
4      @PersistenceContext(unitName="default")
5      private EntityManager entityManager;
6
7      Context ctx;
8      StudentCourse studentCourseEJB;
9      public void ejbCreate() {
10         try {
11             studentCourseEJB = (StudentCourse) ServiceLocator.
getService("java:module/StudentCourseEJB!it.marocco.
maroccoass4_2server.ejb.StudentCourse");
12         } catch (NamingException e) {
13             throw new RuntimeException(e);
14         }
15     }
16
17     @Override
18     public List<TeacherEntity> getStudentTeachers(int
matriculation) {
19         List<StudentCourseEntity> sc = studentCourseEJB.
getStudentCourses(matriculation);
20         if (sc == null) return null;
21         List<TeacherEntity> t = new ArrayList<>();
22         for (StudentCourseEntity s : sc) {
23             t.add(s.getCourse().getTeacher());
24         }
25         return t;
26     }
27 }
28
```

Code 7: TeachersEJB

Since the StudentCourseEntity already has the full referenced object, we do not need a new query: we can use the same one used before and just go a little bit deeper in the resulted object. We start by creating a lookup link to our StudentCourseEJB at the creation of this bean, done in the function ejbCreate, then we call its method and extract from each StudentCourseEntity the referenced TeacherEntity (this is why in the architecture TeacherEJB did not reach out to TeacherEntity directly).

We can see here a strange thing, and another one across all this beans: instead of doing a classic lookup we use something called ServiceLocator, and all our beans are locals! The reason is simple: we are using well established patterns.

6 Patterns

Patterns are a well known and studied way to resolve common problems. Here we show how we are using two of them. We are using two more, DTO and Business delegates, but we will talk about them later when distributing the application.

6.1 ServiceLocator

Context Lookup are heavy in resources to execute: most of the time it may happen that we look up multiple times the same bean. It is much more efficient to delegate all the lookups to a single class which can cache them for multiple calls without much overhead.

```
1  public class ServiceLocator {
2      private static HashMap<String, Object> cache;
3
4      static {
5          cache = new HashMap<String, Object>();
6      }
7
8      public static Object getService(String jndiName) throws
NamingException {
9          Object service = cache.get(jndiName);
10         if (service == null) {
11             InitialContext context = new InitialContext();
12             service = context.lookup(jndiName);
13             cache.put(jndiName, service);
14         }
15         return service;
16     }
17 }
18
```

Code 8: ServiceLocator

6.2 FacadeEJB

It is not ideal having the client interface with multiple beans: what we can do is create a single Remote EJB which proxies all the function we need to expose, so that the client only needs to connect to as few EJB as possible.

```
1      @Stateless
2      @Remote(Facade.class)
3      public class FacadeEJB implements Facade {
4          StudentCourse studentCourseEJB;
5          Teachers teachersEJB;
6          Students studentsEJB;
7          public void ejbCreate() {
8              try {
9                  studentCourseEJB = (StudentCourse) ServiceLocator.
getService("java:module/StudentCourseEJB!it.marrocco.
marroccoass4_2server.ejb.StudentCourse");
10                 teachersEJB = (Teachers) ServiceLocator.getService
("java:module/TeachersEJB!it.marrocco.
marroccoass4_2server.ejb.Teachers");
11                 studentsEJB = (Students) ServiceLocator.getService
("java:module/StudentsEJB!it.marrocco.
marroccoass4_2server.ejb.Students");
12             } catch (NamingException e) {
13                 throw new RuntimeException(e);
14             }
15         }
16
17         @Override
18         public StudentEntity getSingleStudent(int
matriculation) {
19             return studentsEJB.getSingleStudent(matriculation);
20         }
21
22         @Override
23         public List<StudentCourseEntity> getStudentCourses(int
matriculation) {
24             return studentCourseEJB.getStudentCourses(
matriculation);
25         }
26
27         @Override
28         public List<TeacherEntity> getStudentTeachers(int
matriculation) {
29             return teachersEJB.getStudentTeachers(matriculation)
;
30         }
31     }
```

Code 9: FacadeEJB

7 Front End

7.1 Business Delegate

The team working on the frontend needs to have ready to use functions, without having to know how the backend works or how to connect to it. This is why we create a Business Delegate, a class made by the backend team which gives the functions useful for the frontend application.

In our case, the backend team need to give two things. The first is a subset of the package used in the server containing the Facade interface and the Entities classes. This is needed because the backend methods will return these classes, which are just objects with some fields, getters/setters and equals/hashCode functions (cleaned of the extra injections used in the backend). They are implementing Serializable, which makes easy for them to be shared remotely (DTO). To make JNDI work, we need to put these files, which are in common with the server, in the same package structure as the server itself, otherwise the connection will return the `ClassNotFoundException`.¹

The second thing the backend team will give to the frontend one is a folder containing the business delegate class and a service locator. The frontend team will need only to interface with the business delegate class to do their job. The service locator for the client is not used to find the different EJBs inside the project, but to interface with the server: in this case, we need the *InitialContext* to have some properties to tell him where to connect (properties given by the function *getJndiProperties()*). We can see how also here the Service Locator is a Singleton.

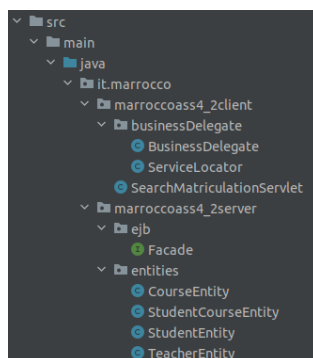


Figure 7.1: The structure of the Client project

¹<https://stackoverflow.com/questions/13701041/remote-ejb-call-class-not-found-exception/13703265>

```

1 public class BusinessDelegate {
2     Facade facadeEJB;
3     public BusinessDelegate() {
4         try {
5             String jndiName = "ejb:/MarroccoAss4_2-Server
-1.0-SNAPSHOT/FacadeEJB!it.marrocco.marroccoass4_2server.
ejb.Facade";
6             facadeEJB = (Facade) ServiceLocator.getService
(jndiName);
7         } catch (NamingException e) {
8             System.out.println("Naming exception: " + e.
getMessage());
9             e.printStackTrace();
10            throw new RuntimeException(e);
11        }
12    }
13
14    public StudentEntity getSingleStudent(int
matriculation) {
15        return facadeEJB.getSingleStudent(matriculation);
16    }
17
18    public List<StudentCourseEntity> getStudentCourses(int
matriculation) {
19        return facadeEJB.getStudentCourses(matriculation);
20    }
21
22    public List<TeacherEntity> getStudentTeachers(int
matriculation) {
23        return facadeEJB.getStudentTeachers(matriculation)
;
24    }
25 }

```

Code 10: BusinessDelegate

```

1 public class ServiceLocator {
2     private static HashMap<String, Object> cache;
3
4     static {
5         cache = new HashMap<String, Object>();
6     }
7
8     private static Properties getJndiProperties() {
9         Properties jndiProperties=new Properties();
10        jndiProperties.put(Context.INITIAL_CONTEXT_FACTORY
11        , "org.wildfly.naming.client.WildFlyInitialContextFactory
12        ");
13        jndiProperties.put(Context.PROVIDER_URL,"http-
14        remoting://localhost:8080");
15        jndiProperties.put(Context.URL_PKG_PREFIXES, "org.
16        jboss.ejb.client.naming");
17        jndiProperties.put("jboss.naming.client.ejb.
18        context", true);
19
20        return jndiProperties;
21    }
22
23    public static Object getService(String jndiName)
24    throws NamingException {
25        Object service = cache.get(jndiName);
26        if (service == null) {
27            InitialContext context = new InitialContext(
28            getJndiProperties());
29            service = context.lookup(jndiName);
30            cache.put(jndiName, service);
31        }
32        return service;
33    }
34 }

```

Code 11: Client ServiceLocator

7.2 Html Pages

First we create the `index.jsp` file, which is a simple form that has a field for the matriculation number and two checks, one for each page, to choose what to get from the server.

The form will redirect to our servlet, which will take the matriculation number from the request body. It will then instantiate the class `BusinessDelegate` made by the backend team in the `init()` method, call its functions to get the data necessary from the server and use it to create HTML parts.

```
1  @WebServlet(name = "searchMatriculation", value = "/"
   searchMatriculation")
2  public class SearchMatriculationServlet extends
   HttpServlet {
3      BusinessDelegate bd;
4
5      @Override
6      public void init() {
7          try {
8              bd = new BusinessDelegate();
9              System.out.println("Connection working");
10         } catch (RuntimeException e) {
11             System.out.println("Cannot get connection to
server");
12         }
13     }
14
15     public String formatStudentEntity(StudentEntity s) {
16         return "<h1>" + s.getSurname() + " " + s.getName()
+ " (" + s.getMatriculation() + ")</h1>";
17     }
18
19     public String getStudentPageElement(int matriculation)
{
20         StudentEntity s = bd.getSingleStudent(
matriculation);
21         if (s == null) return "<h1>Student was not found</
h1>";
22         String html = formatStudentEntity(s);
23         html += "<h2>Courses:</h2>";
24         try {
25             html += "<ul>";
26             List<StudentCourseEntity> sc = bd.
getStudentCourses(matriculation);
27             if (sc == null) return "<h1>Error getting the
Student Courses</h1>";
28             for (StudentCourseEntity c : sc) {
29                 html += "<li>" + c.getCourse().getName();
```



```

30         if(c.getGrade() != null)
31             html += " (grade = " + c.getGrade() +
32         ");
33             html += "</li>";
34         }
35         html += "</ul>";
36     } catch (Exception e ) {
37         System.out.println("error: " + e.getMessage())
38     ;
39         html += "<h3>error<h3>";
40     }
41     return html;
42 }
43
44 public String getAdvisoryPageElement(int matriculation
45 ) {
46     StudentEntity s = bd.getSingleStudent(
47     matriculation);
48     if (s == null) return "<h1>Student was not found</
49     h1>";
50     String html = formatStudentEntity(s);
51     html += "<h2>Advisors:</h2>";
52     try {
53         html += "<ul>";
54         List<TeacherEntity> sc = bd.getStudentTeachers
55         (matriculation);
56         if (sc == null) return "<h1>Error getting the
57         Student Teachers</h1>";
58         for (TeacherEntity t : sc) {
59             html += "<li>" + t.getSurname() + " " + t.
60             getName() + "</li>";
61         }
62         html += "</ul>";
63     } catch (Exception e ) {
64         System.out.println("error: " + e.getMessage())
65     ;
66         html += "<h3>error<h3>";
67     }
68     return html;
69 }
70
71 public void doPost(HttpServletRequest request,
72 HttpServletResponse response) throws IOException {
73     int matriculation;
74     try {
75         matriculation = Integer.parseInt(request.
76         getParameter("matriculation"));
77     } catch (NumberFormatException e) {
78         response.sendRedirect("");
79     }
80 }

```

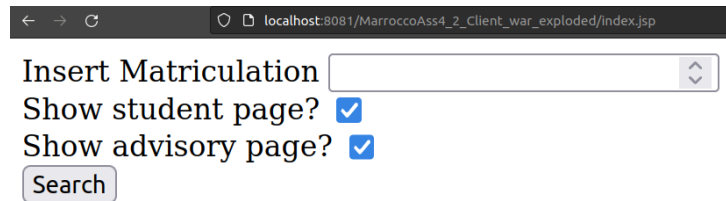
```

68         return;
69     }
70     boolean showStudentPage = request.getParameter("
studentPage") != null;
71     boolean showAdvisoryPage = request.getParameter("
advisoryPage") != null;
72
73     String html = "";
74     try {
75         if (showStudentPage) html +=
getStudentPageElement(matriculation);
76         if (showAdvisoryPage) html +=
getAdvisoryPageElement(matriculation);
77     } catch (Exception e) {
78         System.out.println("Error in fetching data");
79         html += "<h1>Error in fetching data</h1>";
80         html += "<p>" + e.getMessage() + "</p>";
81     }
82
83     response.setContentType("text/html");
84     PrintWriter out = response.getWriter();
85     out.println("<html><head><title>Matriculation "+
matriculation+"</title></head><body>");
86     out.println(html);
87     out.println("<a href='index.jsp'>Go back</a>");
88     out.println("</body></html>");
89     out.close();
90 }
91 }

```

Code 12: SearchMatriculationServlet

7.3 Pages Images

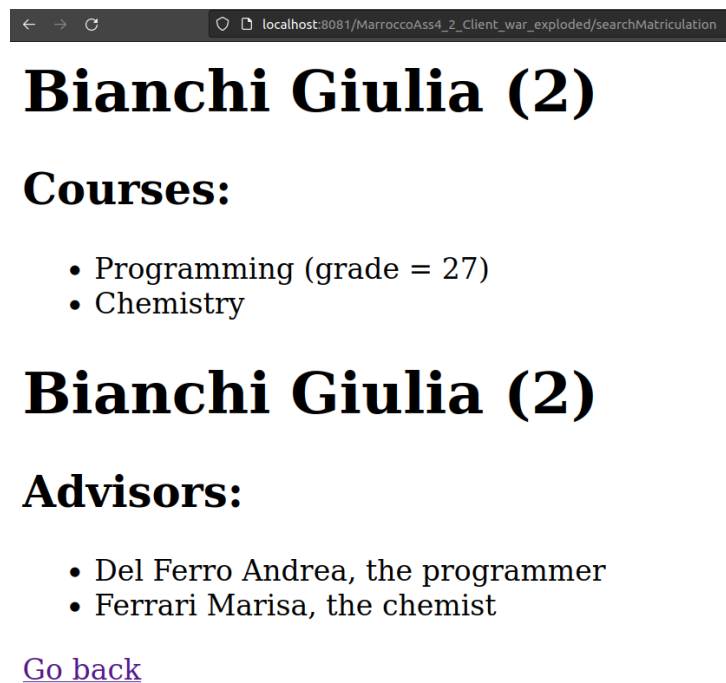


Insert Matriculation

Show student page? ☒

Show advisory page? ☒

Figure 7.2: The starting page



Bianchi Giulia (2)

Courses:

- Programming (grade = 27)
- Chemistry

Bianchi Giulia (2)

Advisors:

- Del Ferro Andrea, the programmer
- Ferrari Marisa, the chemist

[Go back](#)

Figure 7.3: The servlet page when both checks are set, showing both requested pages

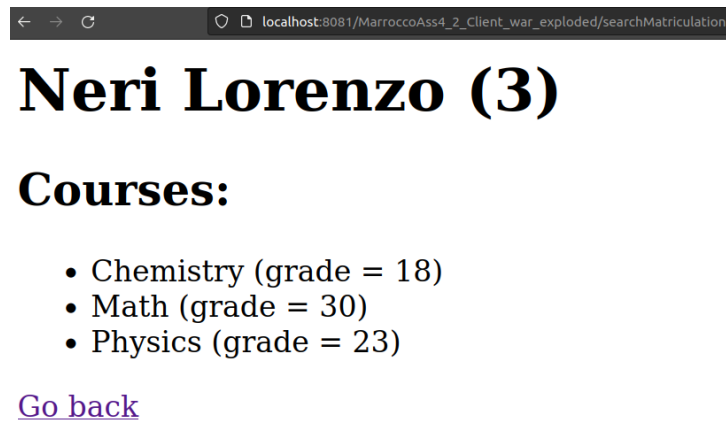


Figure 7.4: The servlet page with only the courses page



Figure 7.5: The servlet page when a non existant student is searched

8 Appendix A: Database Initialization

For better debugging, in the Teachers names strings it was added a short text describing their role. Of course, it is not something that would be added in a real case scenario.

```
1  drop table if exists STUDENT_COURSE;
2  drop table if exists COURSE;
3  drop table if exists STUDENT;
4  drop table if exists TEACHER;
5
6  create table STUDENT (
7      MATRICULATION int NOT NULL AUTO_INCREMENT,
8      NAME varchar(255) NOT NULL,
9      SURNAME varchar(255) NOT NULL,
10
11     primary key (MATRICULATION)
12 );
13 create table TEACHER (
14     MATRICULATION int NOT NULL AUTO_INCREMENT,
15     NAME varchar(255) NOT NULL,
16     SURNAME varchar(255) NOT NULL,
17
18     primary key (MATRICULATION)
19 );
20 create table COURSE (
21     NAME varchar(255) NOT NULL,
22     TEACHER varchar(255) NOT NULL UNIQUE,
23
24     primary key (NAME),
25     foreign key (TEACHER) references TEACHER(MATRICULATION)
26 );
27 create table STUDENT_COURSE (
28     STUDENT_MATRICULATION int NOT NULL,
29     COURSE_NAME varchar(255) NOT NULL,
30     GRADE int,
31
32     primary key (STUDENT_MATRICULATION, COURSE_NAME),
33     foreign key (STUDENT_MATRICULATION) references STUDENT
(MATRICULATION),
34     foreign key (COURSE_NAME) references COURSE(NAME)
35 );
36
37 insert into STUDENT values (1, 'Mario', 'Rossi');
38 insert into STUDENT values (2, 'Giulia', 'Bianchi');
39 insert into STUDENT values (3, 'Lorenzo', 'Neri');
40 insert into STUDENT values (4, 'Rita', 'Verdi');
41
```

```

42     insert into TEACHER values (1, 'Andrea, the programmer',
43     'Del Ferro');
44     insert into TEACHER values (2, 'Giuseppe, the
45     mathematician', 'Benedetti');
46     insert into TEACHER values (3, 'Marisa, the chemist', '
47     Ferrari');
48     insert into TEACHER values (4, 'Lucrezia, the physicist'
49     , 'Bruno');
50
51     insert into COURSE values ('Programming', 1);
52     insert into COURSE values ('Math', 2);
53     insert into COURSE values ('Chemistry', 3);
54     insert into COURSE values ('Physics', 4);
55
56     insert into STUDENT_COURSE values (1, 'Programming',
57     null);
58     insert into STUDENT_COURSE values (1, 'Math', 25);
59     insert into STUDENT_COURSE values (2, 'Programming', 27)
60     ;
61     insert into STUDENT_COURSE values (2, 'Chemistry', null)
62     ;
63     insert into STUDENT_COURSE values (3, 'Chemistry', 18);
64     insert into STUDENT_COURSE values (3, 'Math', 30);
65     insert into STUDENT_COURSE values (3, 'Physics', 23);
66     insert into STUDENT_COURSE values (4, 'Physics', 30);
67
68     select * from STUDENT;
69     select * from TEACHER;
70     select * from COURSE;
71     select * from STUDENT_COURSE;
72
73     select * from STUDENT, TEACHER, COURSE, STUDENT_COURSE
74     where STUDENT.MATRICULATION = STUDENT_COURSE.
75     STUDENT_MATRICULATION
76     and COURSE.NAME = STUDENT_COURSE.COURSE_NAME
77     and COURSE.TEACHER = TEACHER.MATRICULATION;

```

Code 13: Queries to initialize the DB