UNIVERSITÀ DI PISA

MSc in Computer Engineering

Distributed Systems and Middleware Technologies

University Student Platform

TEAM MEMBERS:

Fabrizio Lanzillo
Federico Montini
Riccardo Sagramoni

https://github.com/FabrizioLanzillo/Distributed-Students-Chat

---

Academic Year 2022-2023

Contents

Chapter 1

Project Specifications

1.1 Introduction

Student University Platform is a distributed web-app aimed at providing various functionalities to students and professors, which are:

- allow booking and managing of meetings
- provide chatroom functionality among students where they can share opinions and suggestions

Both students and professors have a dedicated page where they can view scheduled meetings and cancel them. Within the course page, on the other hand, it is possible to access the dedicated chatroom or look at the list of available slots and choose the meeting to reserve.

1.2 Functional and Non-functional Requirements

In our web-application, four types of actors exist:

1. Unregistered student
2. Students
3. Professors
4. Admin

The operation that each kind of actor can perform in the application are the following.

1. An **unregistered student** can:
    - create an account
2. A **student** can:
    - login/logout
    - browse courses by course name

    - see his starred course
    - see and manage his booked meetings
    - select a course in order to:
        - view course details
        - star that course
        - enter the course chatroom
        - book a meeting for that course
3. A **professor** can:
    - login/logout
    - see and manage his booked meetings
    - create a new course selecting specific time-slots
    - delete a course
4. The **admin** can:
    - login/logout
    - create professor account
    - browse users by username and ban them

For what concerns **non-functional requirements** we have, **for the web-app**:

- Concurrent service accesses management
- Strong consistency for users, courses and meetings data stored in MySQL DB

and, **for the chatroom**:

- Concurrent service accesses management
- High service availability
- Fault tolerance to node faults
- Allow high horizontal scalability
- Eventual consistency for chatroom state data

1.3 Synchronization and communication issues

Let's introduce briefly the main communication and synchronization issues:

- Multiple clients exchange messages with each other in chat rooms concurrently. The chat of each client node must be synchronized.
- When a student enters or leave the chatroom each client node must be synchronized in order to see a consistent list of students within the chat (i.e. "online students").
- When a student books a meeting in an available time slot or remove a previously booked meeting, each client node must be synchronized to view the same consistent state (both for students and professors).
- When a professor creates or delete a course, the server must show the course to all the connected client in the "browse", "search" sections and, in case of deletion, inside the course page.

Chapter 2

System Architecture

2.1 Introduction

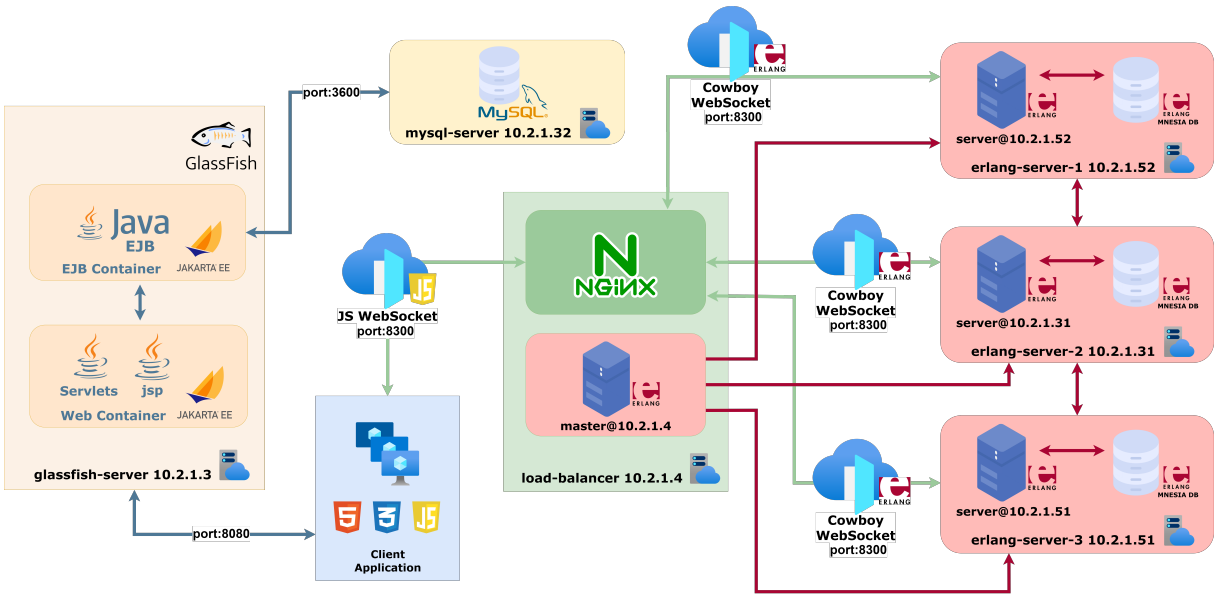

Figure 2.1: System Architecture Schema.

The web application, as we can see in figure 2.1, consists of a **Client Application**, a **Jakarta Application Server**, and **multiple erlang servers**.
The Jakarta EE Application Server, which implements most of the functionality of the application, uses as its reference implementation, **Glassfish 6.2.5** and also communicates with a **MySql relational database** located on a different machine.
The Erlang Servers, which are managed by a **Load Balancer** on another machine, handle the Chatrooms of the application through an **HTTP websocket communication**.
For more specific details, all these modules will be analyzed individually.

2.2 Erlang

Within our project, we developed the **chatroom** functionality in *Erlang*.

In order to take full advantage of Erlang's innate support for concurrency and scalability and to meet out requirements for high availability and fault tolerance, we deployed the chatroom in a **distributed** manner, on *multiple "server" nodes*. Specifically for this project, we configured three different containers to run their own Erlang instance (`10.2.1.31`, `10.2.1.51`, `10.2.1.52`).

Each individual node exposes an endpoint to connect to the users' browsers via the **WebSocket** protocol. In this way, the server can receive chat messages from a user and forward them to other online users, meanwhile the user is not required to refresh the webpage.

The Erlang-side WebSocket communication been implemented by leveraging **Cowboy**, a small, fast and modern HTTP server for Erlang/OTP. This allowed us to focus only on the business logic and the remote deployment, without having to deal with low level HTTP protocols. On the other hand, the client-side WebSocket communication has been implement in **Javascript**. Every message exchanged between client and server is serialized in **JSON format**. The `jsone` library has been used for the Erlang implementation.

Distributed Erlang nodes share chat and user status information through **Mnesia**, a distributed database, integrated by default into Erlang/OTP and based on ETS (in-memory) and DETS (on-disk) as storage mechanisms.

A **load balancer** (Nginx) has been also deployed to fairly distribute the incoming WebSocket requests to the available Erlang servers.

The container hosting the load balancer (`10.2.1.4`) also runs an Erlang process (*master node*) responsible for configuring the Mnesia cluster and spawning the chat server application processes on the remote Erlang nodes.

Finally, **rebar3** has been used as building tool for the Erlang application.

2.2.1 Chat server

Overview

The chat server application is implemented by exploiting the supervisor behavior: at startup, the application process spawns a supervisor process (callback module `chat_server_sup`) which will be responsible to manage the lifecycle of the process which runs the **Cowboy HTTP server**.

The process running Cowboy is implemented as a **gen_server** using the module `cowboy_listener` as callback module. The Cowboy server will listen at the port **8300** waiting for new WebSocket connection requests and it will spawn a new ad hoc process to handle the request once received. Thus, every user will be assigned to a different Erlang process (actually, every user *connection* since a student can open more than one WebSocket connections).

The new spawned process will call the functions belonging to the `chatroom_websocket` module:

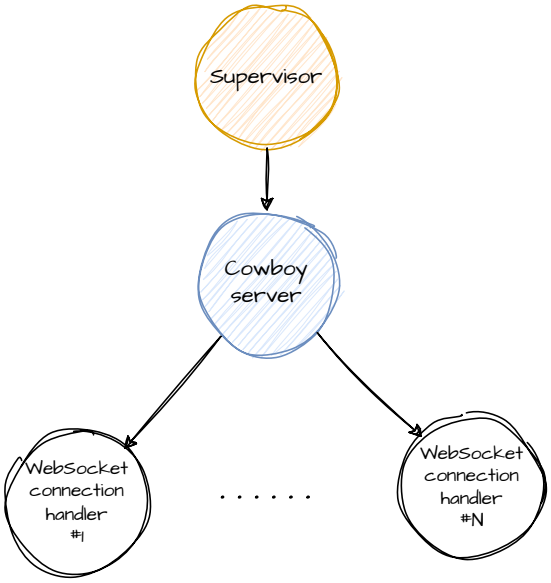

Figure 2.2: Hierarchy of Erlang processes inside the chat server application

- `init/1`: called whenever a request is received, in order to establish a websocket connection.
- `websocket_handle/2`: called whenever a text, binary, ping or pong frame arrives from the client. It will handle the reception and deserialization of the JSON objects coming from the client.
- `websocket_info/2`: called whenever an Erlang message arrives (i.e. a message from another Erlang process). It will forward a chatroom message to the client browser assigned to this process.
- `terminate/3`: called whenever the WebSocket connection is closed. It will remove its assigned user from the list of the currently online students inside the chatroom.

Client-server communication

During the WebSocket session, all messages exchanged between the *client browser* and the *Cowboy WebSocket handler* are encoded in **JSON format**. Meanwhile Javascript natively supports JSON encoding/decoding inside the browser, the Erlang application requires an external dependency such as the **jsone library**.

The client browser can send three different kinds of JSON messages:

- Request for login into a chatroom
- Request for the list of the currently online students in the chatroom, i.e. the students currently logged in the chatroom
- Chat message

The Javascript client encodes these message in the following format:

```
{
    opcode: "LOGIN",
    course: <id of the course owning the chatroom>,
    username: <username of the current user>
}

{
    opcode: "GET_ONLINE_USERS"
}

{
    opcode: "MESSAGE",
    text: <text of message>
}
```

On the other hand, the Cowboy server can send back two different kinds of JSON messages:

- Current list of the currently online students in the chatroom
- Chat message

The encoded format is the following:

```
{
    opcode: "MESSAGE",
    sender: <sender's username>,
    text: <text of message>
}

{
    opcode: "GET_ONLINE_USERS",
    list: <list of currently online users>
}
```

Modules overview

The `chat_server` application is composed of six modules:

- `chat_server_app`: the first executed module. It starts the mnesia application and spawns the supervisor process. It implements the `application` behavior.
- `chat_server_sup`: the `supervisor` callback module. It's responsible to spawn a process which will handle the lifecycle of the process running the Cowboy server. It implements the `supervisor` behavior.
- `cowboy_listener`: a `gen_server` callback module which will start a Cowboy instance listening at port 8300 for TCP connections. It will also cleanup the Mnesia database by removing all the users assigned to this Erlang node which remained after an unexpected crash of the server.
- `chatroom_websocket`: callback module for the module `cowboy_websocket`. For more details see section 2.2.1 on page 6.

- `chatroom_server`: module which implements the business logic for the WebSocket handlers.
- `mnesia_manager`: module which offers methods to access the Mnesia database. For more details see section 2.2.2.

2.2.2 Mnesia Database

Mnesia is a true **distributed DBMS** that we used to store chatrooms informations in a table.
These informations were **distributed over all the erlang nodes**, using *ram_copies* as a **storage option**.
How explained in the **Mnesia documentation**, the *ram_copies* storage option allows us to specify a list of nodes where this table is supposed to have **RAM copies**. A table replica of type *ram_copies* is **not written to disc** on a per transaction basis.
A **table with the cluster schema** of the Erlang nodes, however, **is stored on the disk** of the master node, this schema will be used by the master to identify the cluster nodes on which to run the distributed Mnesia DB.

As previously anticipated, there is **only one table in the database**, ***online_students***, whose type is bag.
The attributes of this table are *course_id, student_pid, student_name* and *hostname*. course_id is the key of this table.
The data in this table is used to **keep track**, for each course, of all **students that join the chatrooms**.
Through the use of different queries, on this table, it is possible to retrieve the online student at any given time in order to send them messages and it is also possible to handle other needs to better manage the chatrooms behavior.

More in detail, the queries we implemented are:

- *join_course*, which **adds a student** to a course chatroom;
- *get_online_pid*, which returns the pids of students currently online in a chatroom, which is a particularly useful query for figuring out **to which erlang processes forward messages**;
- *get_online_students*, which **returns the usernames of students currently online in a chatroom**, these usernames will then be returned to the client in order to show the online students;
- *logout*, which **removes a student** from the related course chatroom;
- *remove_logged_students_by_hostname*, that **recursively removes all users** in each chatroom, **from a given node**.
  In fact, this last query allows us to **guarantee a state of eventual consistency after a node crash**, which would **otherwise introduce duplicates when it is restarted**.

2.2.3 Client-side: Javascript WebSocket

As we anticipated in the section 2.2.1 on page 7. there is a message exchange between the browser client and the cowboy websocket handler.
The browser client sends and receives messages via a **websocket** that is implemented through **Javascipt**.
This websocket was created by using the **WebSocket API**, which provides methods and events to handle websocket connections.
We matched these methods and events with functions that handled them and included additional functions to make the chatroom work properly.

The functions that invoke the **creation and closure of the websocket** are:

- *connect(_logged_user, _course)*, which is responsible for **creating the websocket and associating each method with the function that handles it**. This function is **called with the onload** of the body of the corresponding html page;
- *disconnect()*, which is responsible for **invoking the websocket connection closure** and the function that handles the latter. This function is **called with the onunload** of the body of the corresponding html page.

The functions that **handle the websocket connection** are:

- *openWebsocketConnection()*, it **starts the connection with websocket**, sending the login message and also call of the refreshWebsocketConnection() function;
- *closeWebsocketConnection()*, is invoked when the **connection with the websocket is closed** and also call of the stopRefreshWebsocketConnection(). In addition This function tries to create a new websocket connection **in case the previous connection is involuntarily closed**, for example, after the Erlang Node, to which it was connected, **crashes**;
- *refreshWebsocketConnection()*, it executes an **automatic refresh of the websocket connection** every 10 sec by sending GET_ONLINE_USERS message through the websocket connection;
- *stopRefreshWebsocketConnection()*, it **stops the automatic refresh of the websocket connection**;
- *sendObjectThroughWebsocket(isWebsocketConfigurationMessage, ...args)*,
  it **creates and sends a JSON object** through the websocket connection;
- *receiveObjectThroughWebsocket(event)*, it **handles the receive event of a JSON object**, parse it and executes the action provided by the opcode parameter of the object;
- *handleWebsocketError()*, it **handles the possible errors** with the websocket connection.

2.2.4 Master node

In order to correctly configure the Mnesia database and launch the remote Erlang nodes , we developed a *master application* which executes the following operations:

- It connects to the remote nodes by means of `net_kernel:connect_node/1`.
- It starts its local Mnesia process, creates a persistent schema with the other Erlang nodes (if it doesn't already exist) and creates the *online_students* table inside the schema (if it doesn't already exist). The new table only resides on the chat_server nodes.
- It spawns the chat_server application processes on the remote nodes.

The master application process runs on the same node as the Nginx load balancer.

2.2.5 Load balancer

In order to fairly distribute the load relative to the chatroom and fully exploits the benefits deriving from a distributed Erlang deploy (such as high availability, fault tolerance, scalability...), a *load balancer* has been deployed on node `10.2.1.4`.

We chose **Nginx** for this purpose since it natively supports the WebSocket protocol both as **reverse proxy** and **load balancer**, so it's a perfect match for our requirements.

The Nginx server has been configured to listens on port 8300 and to proxy every WebSocket connection to one of the remote chat_server nodes.

```
worker_processes 1;
worker_rlimit_nofile 8192;

events {
  worker_connections  1024;
}

http {
    map $http_upgrade $connection_upgrade {
        default upgrade;
        '' close;
    }

    upstream websocket {
        server 10.2.1.52:8300;
        server 10.2.1.31:8300;
        server 10.2.1.51:8300;
    }

    server {
        listen 8300;
        location / {
            proxy_pass http://websocket;
            proxy_http_version 1.1;
```

```
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection $connection_upgrade;
            proxy_set_header Host $host;
        }
    }
}
```

2.3 Glassfish

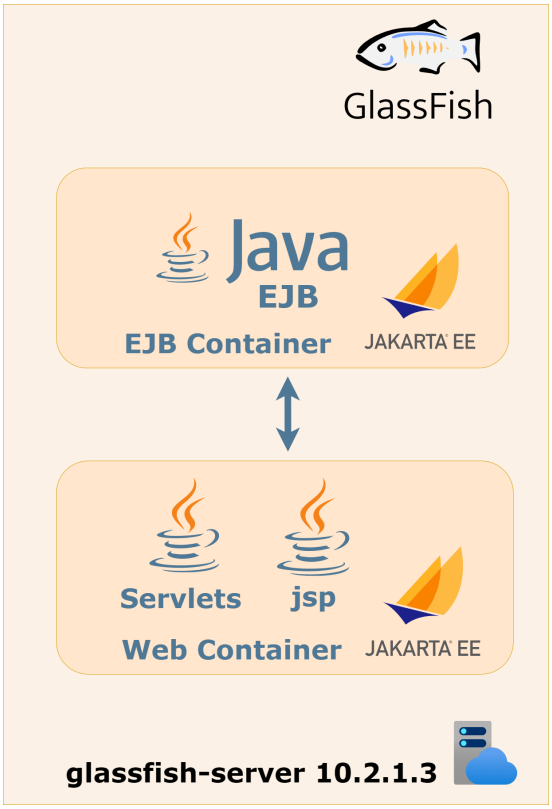

Figure 2.3: Glassfish-Server Schema.

In our project we have implemented the web-app functionality through the Java EE application server known as Glasshfish. This part of the project was divided into three sub-projects in order to separate the business logic from the others application components:

- **web project**: It contains servlets and Java server Pages (JSP). It is responsible for receiving requests and perform calls to EJBs in order to build the requested resources through JSP technology
- **ejb-interfaces**: It contains the interfaces for the calls to EJBs from Sevlets, the structure of all the DTOs and enumerated types
- **ejb**: It contains the implementation of all EJBs and DAOs. EJB are needed to implement all the business logic instead, DAOs, provide an interface between the application and the application's database.

2.3.1 Web

In the Maven project **web**, as said before, we introduced the Servlets and JSP needed. In order to separate each page and keep the logic of each web page divided we implemented a Servlet for each page that will gather the data needed to build the HTML page through JSP technology. In addition, this project contains all the necessary part for displaying resources, so: CSS, JavaScript and the images, everything can be found inside the webapp/assets folder.
Servlets are divided in four main categories based on the type of users to which are dedicated:

- Professor's servlets
- Student's servlets
- Admin's servlets
- Common servlets

Each servlet, that don't belong to common category, access session information to check:

- if the current user has already logged in: otherwise will redirected to login page
- whether the user is in the appropriate category to access that type of resource: otherwise the malicious user will be redirected to his portal page

Common servlets instead, such as login and logout, don't need these kind of techniques since there is no difference between the resource behaviour and the user category.
In the end each servlet, upon a request, can call the methods of the EJBs that implement the related application logic obtaining their remote reference through JNDI.
In the table 2.1 there is a list of the Servlets that has been implemented.

Table 2.1: Servlet List

Visibility	**Servlet Name**	**Implemented methods**
All	LoginServlet	POST
All	LogoutServlet	GET & POST
Students	SignUpServlet	GET
Students	StudentPortalServlet	GET
Students	ChatroomServlet	GET
Students	StudentProfileServlet	GET & POST
Students	StudentCourseServlet	GET & POST
Students	StudentBookingServlet	GET & POST
Professor	ProfessorPortalServlet	GET
Professor	ProfessorMeetingServlet	GET & POST
Professor	ProfessorDeleteCourseServlet	GET & POST
Professor	ProfessorCreateCourseServlet	GET & POST
Admin	AdminCreateProfessorServlet	GET & POST
Admin	AdminPortalServlet	GET
Admin	AdminUserManagementServlet	GET & POST

2.3.2 Ejb-Interfaces

In the **ejb-interfaces** Maven project, as said before, there are **all the interfaces, for the calls to EJBs**.
These interfaces represents the **contract with the client** and they are all ***remote*** interfaces.
In this Maven project there are also all the DTOs and the enumerates type, that will be discussed in the POJOs section.

The interfaces that we have implemented are:

- **BookedMeetingEJB**;
- **CourseEJB**;
- **UserEJB**.

2.3.3 Ejb

The **ejb** Maven project, as previously introduced, contains the **implementation of the classes declared in the ejb-interfaces** with the **business logic needed** to provide the required functionalities.
The type of the **EJB class implementation** is ***Stateless***.
As we anticipated in the ejb-interfaces section, three ejb were used to implement the business logic of the application:

- **BookedMeetingEJBImpl**, this EJB is the **only one where a DAO was created** for, as it provided business logic **beyond simply accessing database resources**. In fact, this EJB, in addition to retrieving data back from the database, through its DAO methods, grants us the ability to view the entire schedule of a professor's slots and perform various operations on it;
- **CourseEJBImpl**, This EJB allows us to **access course data**, it gives us the ability to search through course names and professors, but also to **perform all these operations applied only to the starred courses** of the specific student.
  We can also insert or delete courses, as well as promote or retract them from the starred status;
- **UserEJBImpl**, this EJB gives us the ability to manage the **business logic of the login and of the signup** of the students and their interaction with the db. It also provides operations that can **alter a user's status**, through banning, or operations such as **create professor and student search**.

2.3.4 POJOs: DTOs, Enums and DAOs

In the Maven projects **ejb-interfaces** and **ejb**, we had to introduce some *Plain Old Java Objects* (**POJOs**) in order to efficiently share data between EJBs and Servlets and to separate different concerns:

- Package `dto` (project *ejb-interfaces*): contains all the **Data Transfer Objects** (DTOs) necessary to transfer information from the business logic tier to the presentation tier. The chosen approach was to develop a separate DTO for every single information, in order to minimize the transferred data and to better separate each concern.
- Package `enums` (project *ejb-interfaces*): contains the *UserRole* enum class, needed to correctly identify the role of an user (student, professor, admin).
- Package `dao` (project *ejb*): contains the **Data Access Object** (DAO) responsible for all the data accesses concerning the booking of meetings. The reason only one DAO has been implemented stems from the fact that BookedMeetinEJB is the only EJB that provides a business logic beyond the simple database access, by manipulating data to make the low-level implementation transparent with regards to the presentation tier.

2.4 MySQL Server

Finally, in order to ensure consistency, permanence and availability of data, MySQL server was used. The final structure of our database can be observed in figure 2.4 and consists of the following tables:

- **Admin**: Table intended to contain the information necessary to allow the admin to log in. It is not linked to any other table since there is no specific information to be maintained for this category of user
- **Student**: Table intended to contain all the information needed for student login plus some personal information needed to identify the student
- **Professor**: Table intended to contain all the information needed for the professor's login plus some personal information needed to identify the professor
- **Course**: Table representing the course entity, has as foreign key the id of the professor who owns the course
- **Meeting_slot**: Table intended to contain the list of weekly slots the course makes available to students. The only foreign key is used to identify the course that has made that slot available
- **Booked_meeting**: Table intended to contain the list of slots booked by a student. It has two foreign keys, one needed to identify the student who booked that meeting and the other that binds the meeting to the slot of the course
- **Student_starred_course**: Table needed to keep in memory the set of courses that each student has added to favorites. It has two foreign keys, the combination of which constitutes the primary key, corresponding to the id of the student who added the course to the favorites and the id of the course

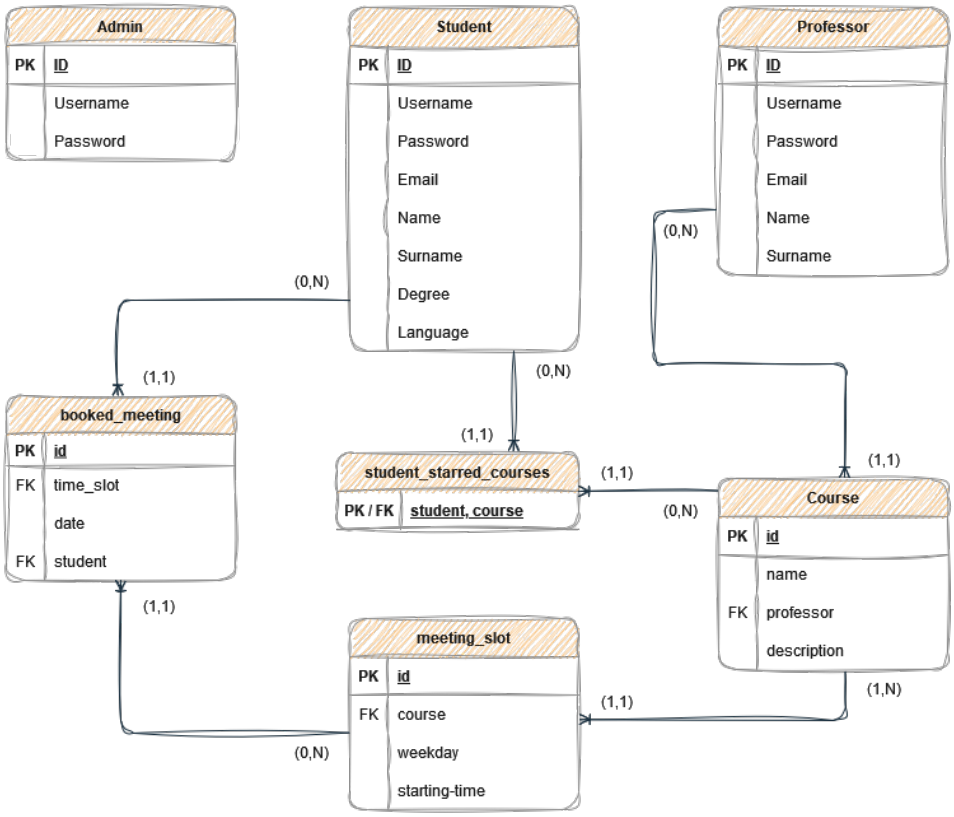

Figure 2.4: Entity-Relation diagram of MySQL DB

Chapter 3

User Manual

3.1 Login page

The root page of the webapp allows a user to login into the platform, by enterin their username, password and role (student, professor or admin).

It's also available a button for unregistered students which redirects them to the sign-up page.

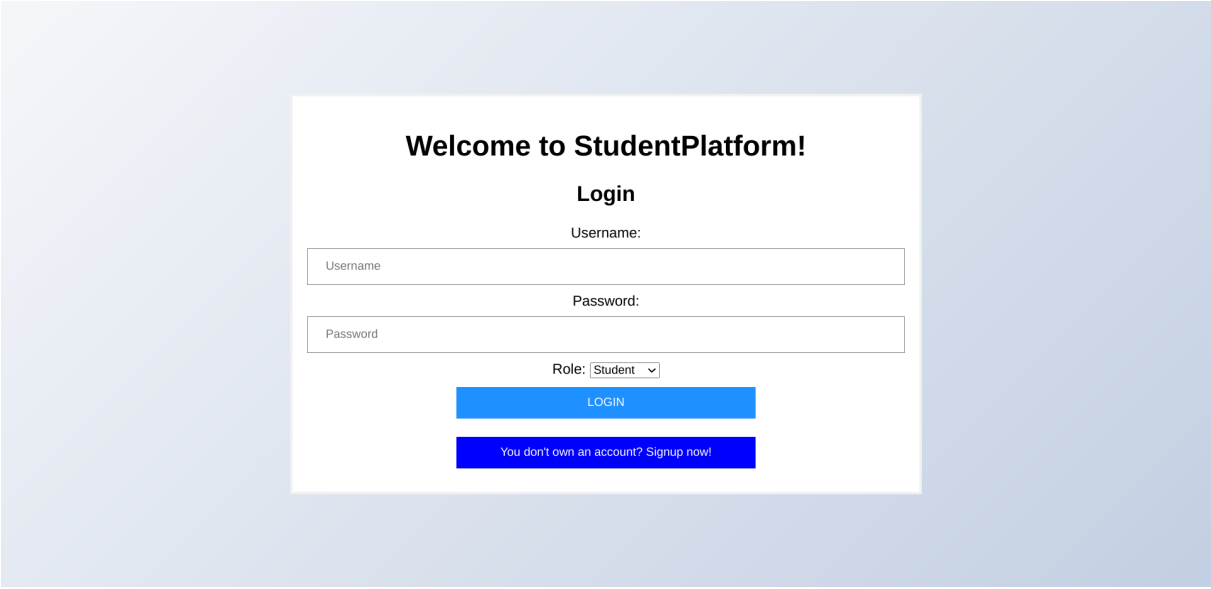

Figure 3.1: Screenshot of login page

3.2 Sign-up page

This page allow unregistered users to sign up to the platform.
It can be accessed from login page and shows a list of fields that the user is required to fill with its personal information in order to sign up. There are two buttons at the bottom of the page: the first allows the user to sign up for the platform, the second allows the user to return to the login page.
After that the first button has been clicked if the sign-up operation has been successful the user will get redirected to login page, otherwise an error message will appear.

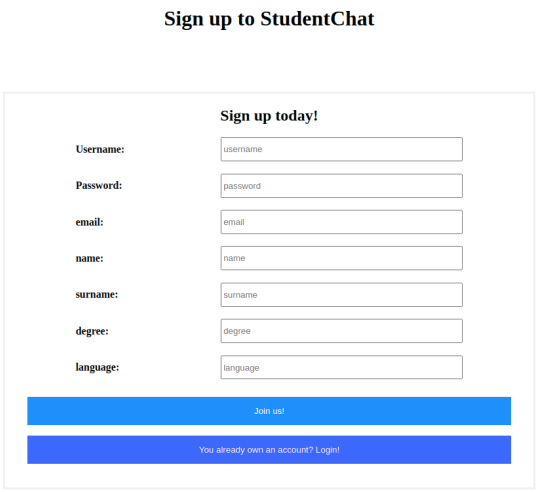

Figure 3.2: Screenshot of sign-up page

3.3 Student section

3.3.1 Portal Page

This page shows the student portal.

The portal displays all the courses that are on the platform.
The student can search for the courses he is interested in, using the research bar, searching by course name or by professor's last name.
He can also view only the courses he has marked as starred.
To access a course, simply click on the blue box of the chosen course.

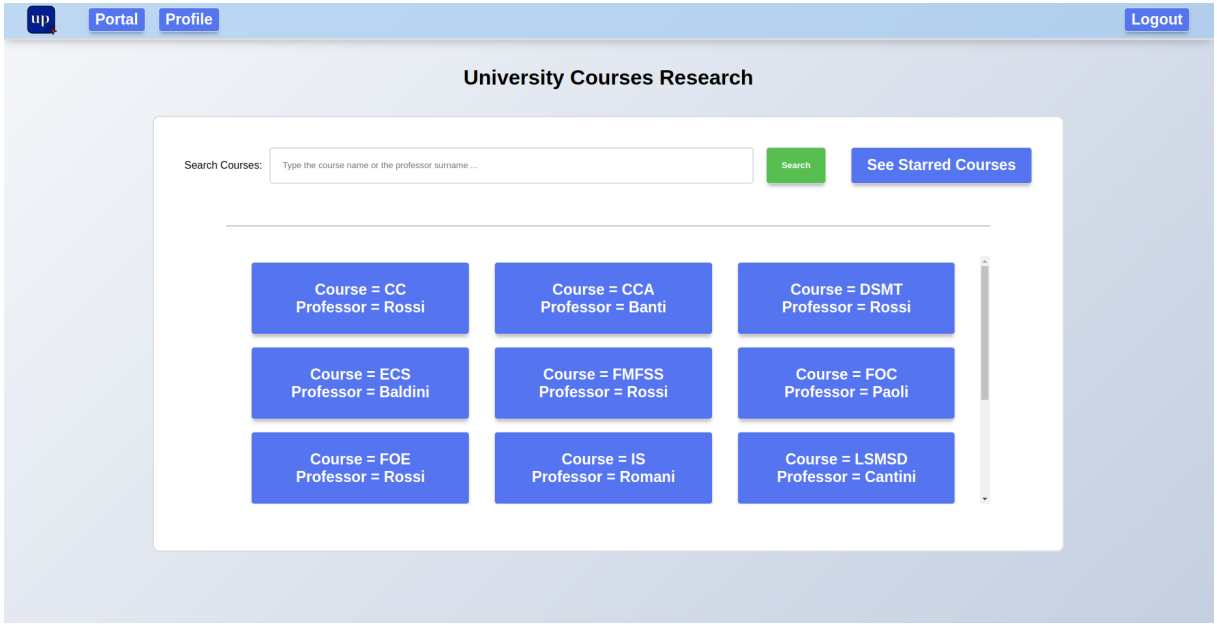

Figure 3.3: Screenshot of the student's portal.

As we can see from the figure, there is a topbar at the top of the page, this topbar is also present in all the other student pages.
Through this topbar we can return to this page, view the student's profile or log out.

3.3.2 Course page

This page shows the details of a course.

It also provides buttons to access the corresponding chatroom and book a meeting with the professor.

Figure 3.4: Screenshot of a course page

Moreover, it is possible for a student to star the course, so that it can be shown in a separate section on their own portal page. When a student clicks on the button *Start this course*, an alert pop-up spawns to notify the success of the request. The same pattern applies for the removal of a star (*unstar*).

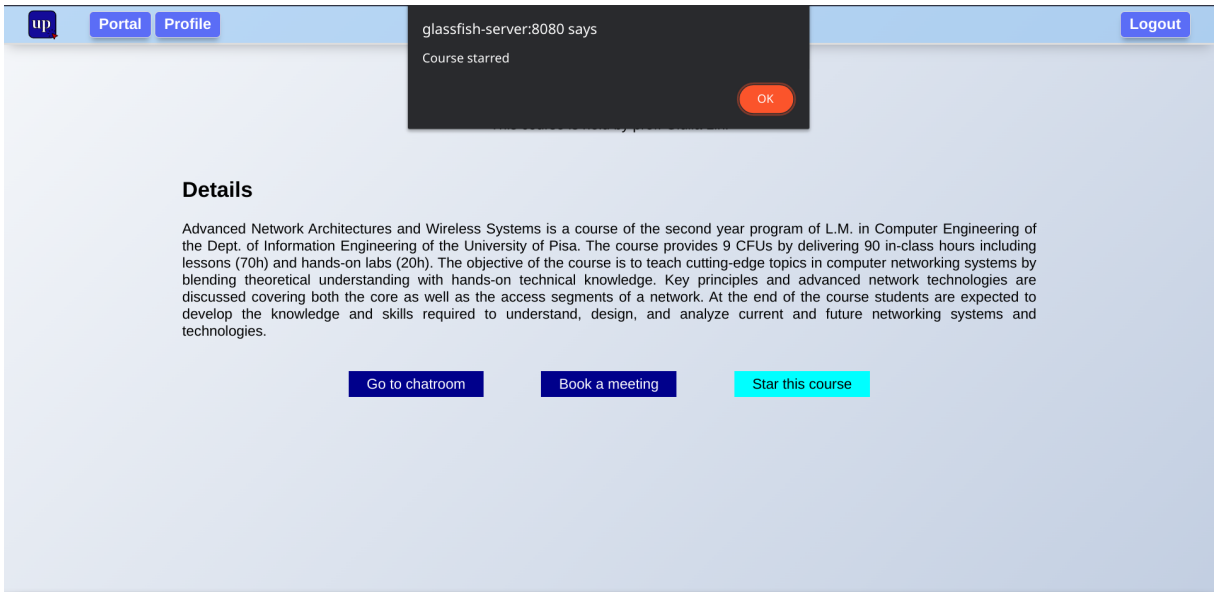

Figure 3.5: Screenshot of a course page showing the alert pop-up which notify that *the course has been successfully starred*

3.3.3 Chatroom page

This is the chatroom page for a specific course.
On this page it is possible to chat with other students who are connected to the same chatroom of the same course.
A student can:

- View the students currently online in the chatroom;
- Send messages in the chatroom through the provided input field;
- View messages sent by other students.

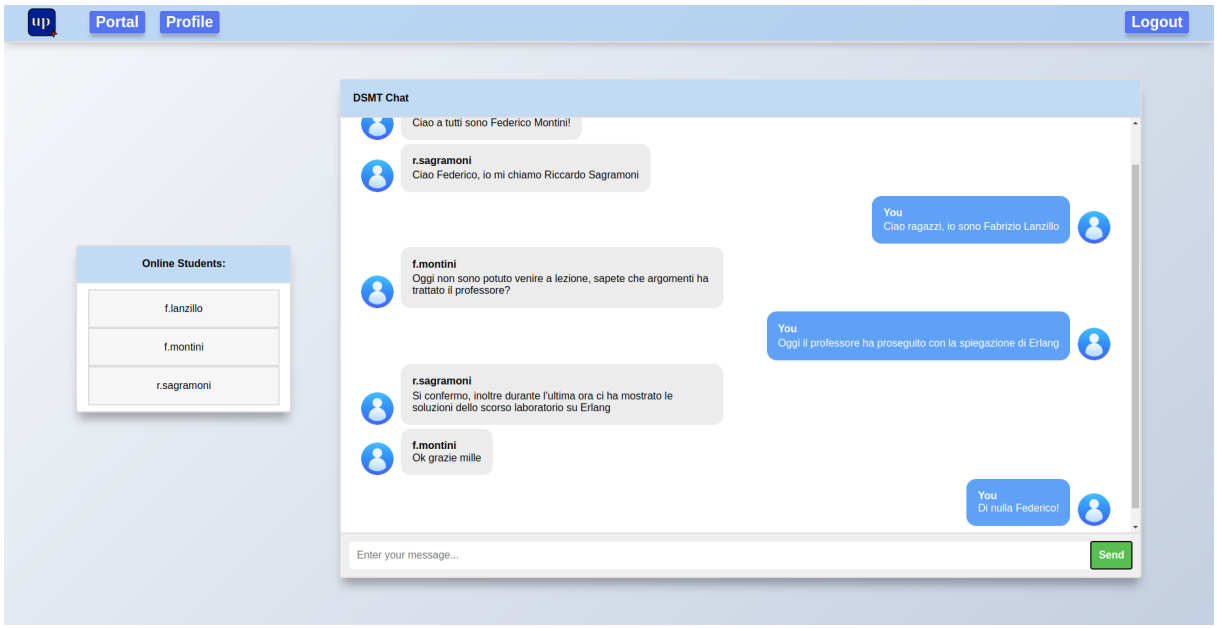

Figure 3.6: Screenshot of the student chatroom page.

3.3.4 Booking page

The page shows the slots in which the student can book a meeting with the course's professor.
The student, in this page, can both book a meeting clicking on the chosen slot or switch month in which he wants to book.

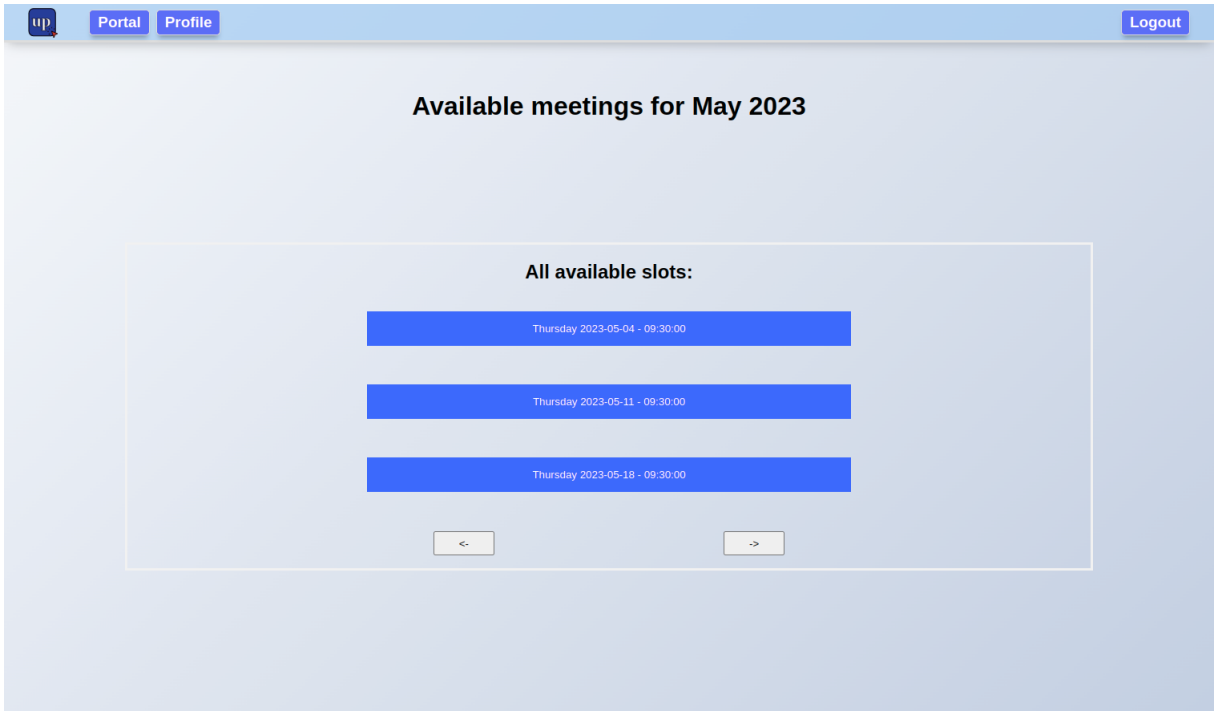

Figure 3.7: Screenshot of student/booking page

Booking a slot the pages shows a message stating the result of the operation.

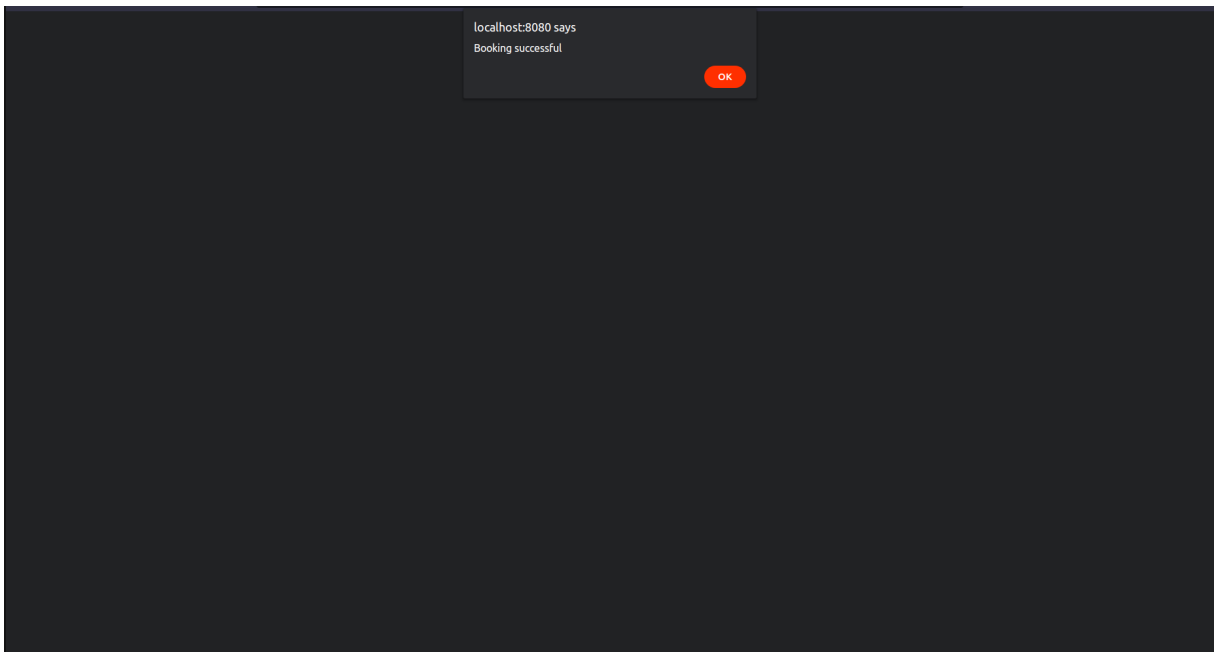

Figure 3.8: Screenshot of booking response

3.3.5 Profile Page

This is the student profile page, accessible only from the topbar.
Here the student can view all his/her meetings that he/she has booked.
If the student wishes to delete a meeting, simply click on the blue box of the corresponding meeting.

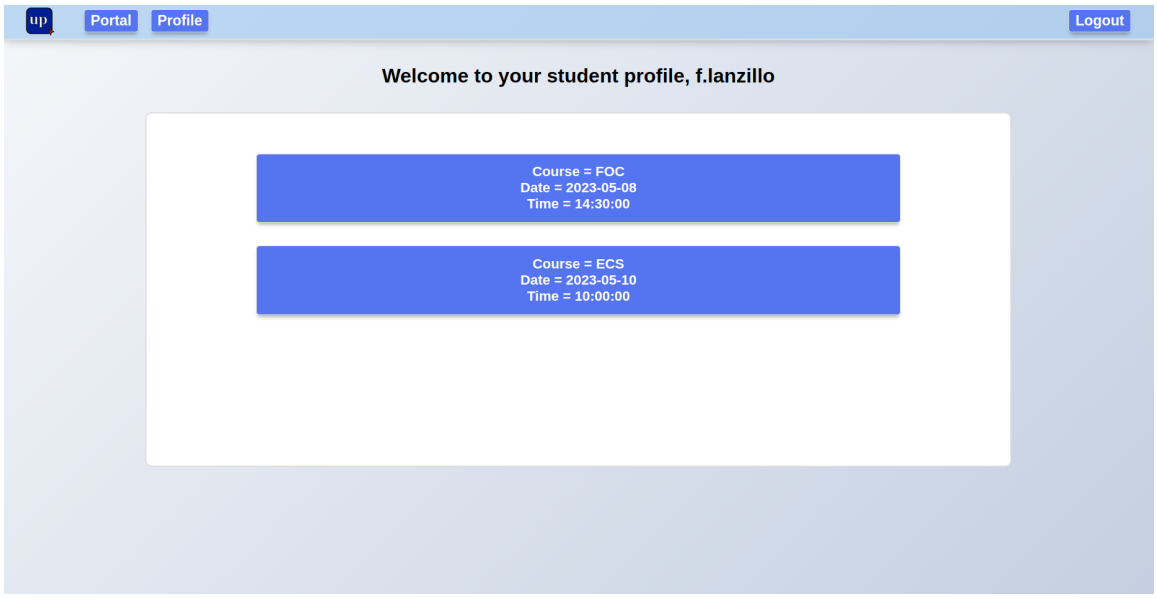

Figure 3.9: Screenshot of the student profile page.

Immediately after selecting the meeting to be deleted, an alert pops up and asks for the confirmation for the delete.
If the student wants to confirm he/she must click on the delete button otherwise on the x.

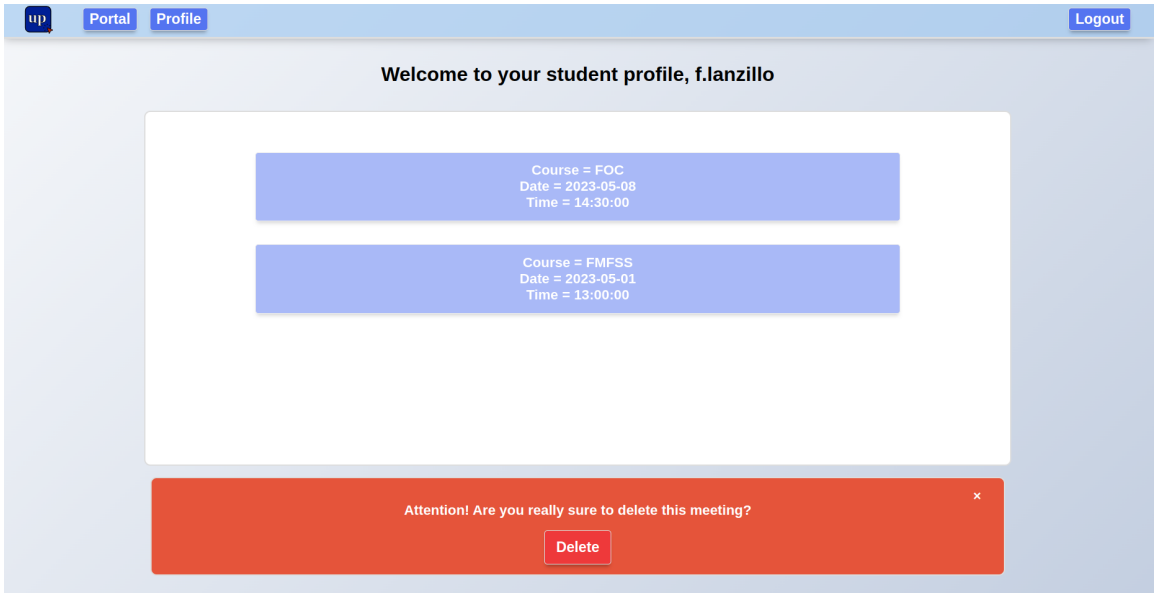

Figure 3.10: Screenshot of the delete alert of the student profile page.

The page shows a message with the result of the delete operation.

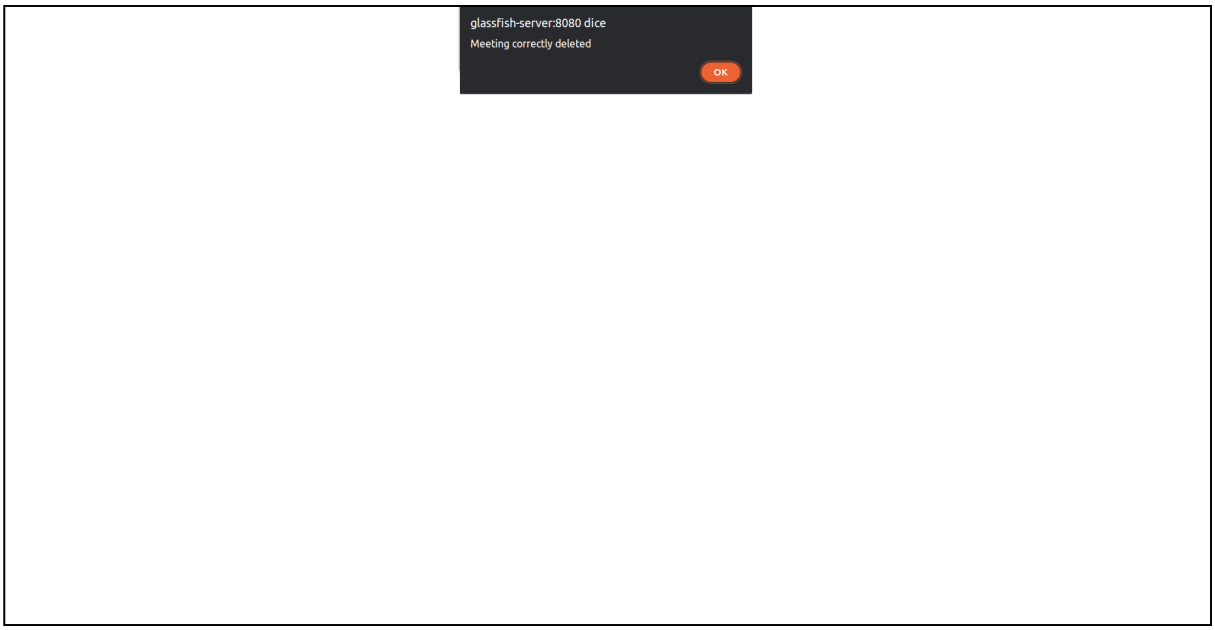

Figure 3.11: Screenshot of the student profile page after the successful delete of the meeting.

3.4 Professor section

3.4.1 Portal page

The portal page for professors provides buttons for the three main operations allowed to professors:

- Creating a course.
- Deleting a course belonging to them.
- Checking all their upcoming scheduled meetings with students.

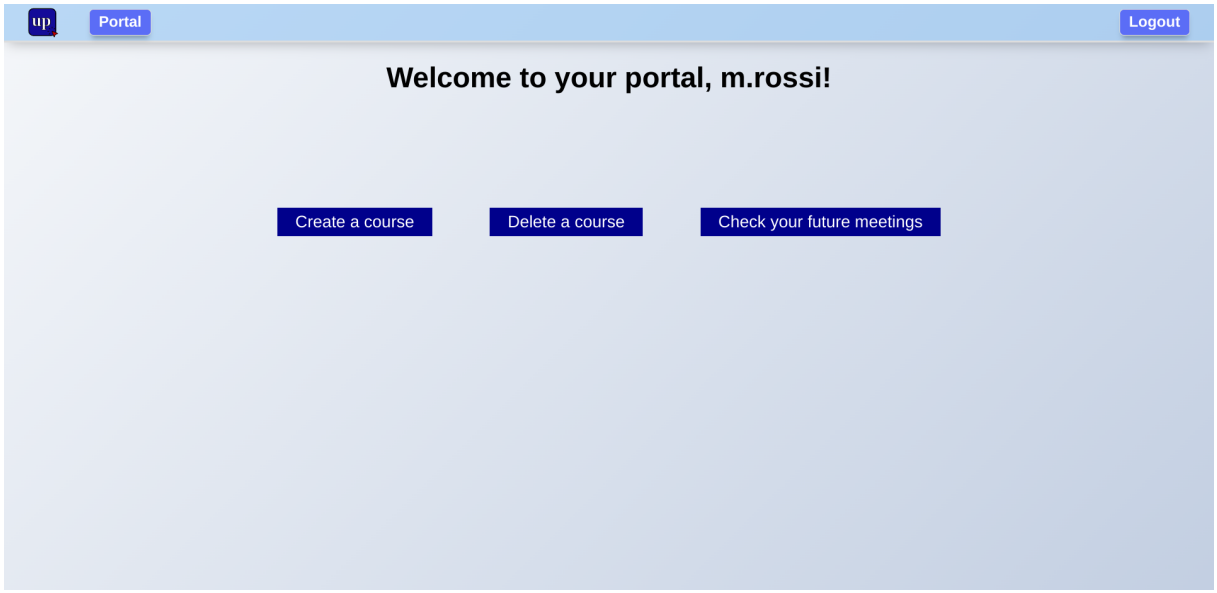

Figure 3.12: Screenshot of the professor's portal

3.4.2 Create a course

This page provides a form to create a new course. Professors can enter the course name and details, and they can specify the weekly time slots for meeting with students.

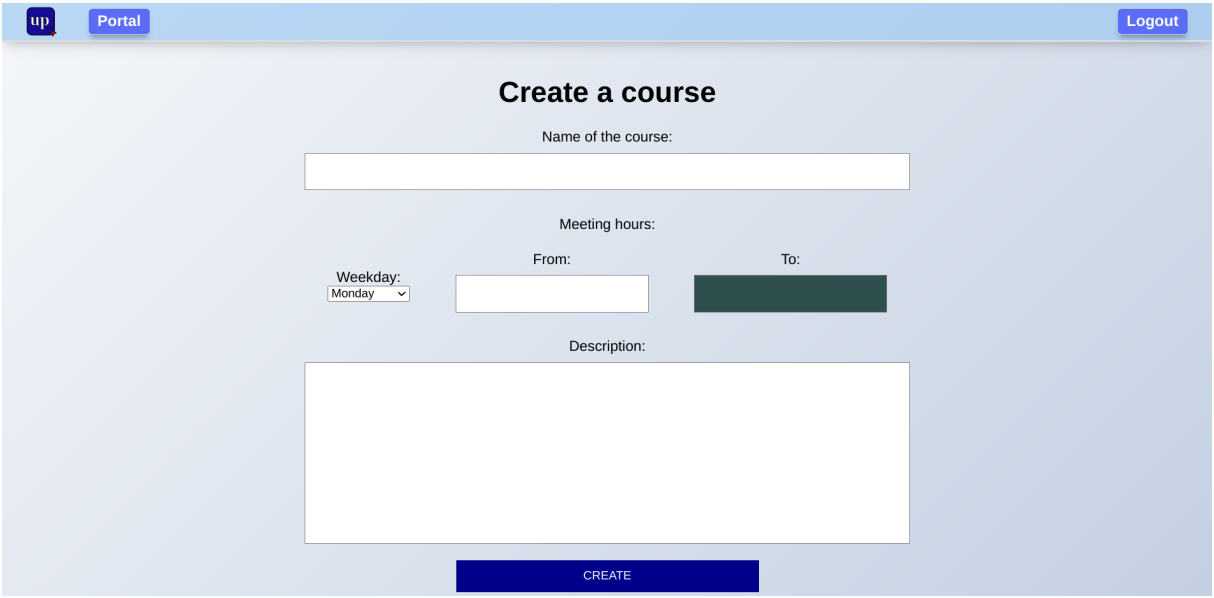

Figure 3.13: Screenshot of the course creation page, with an empty form

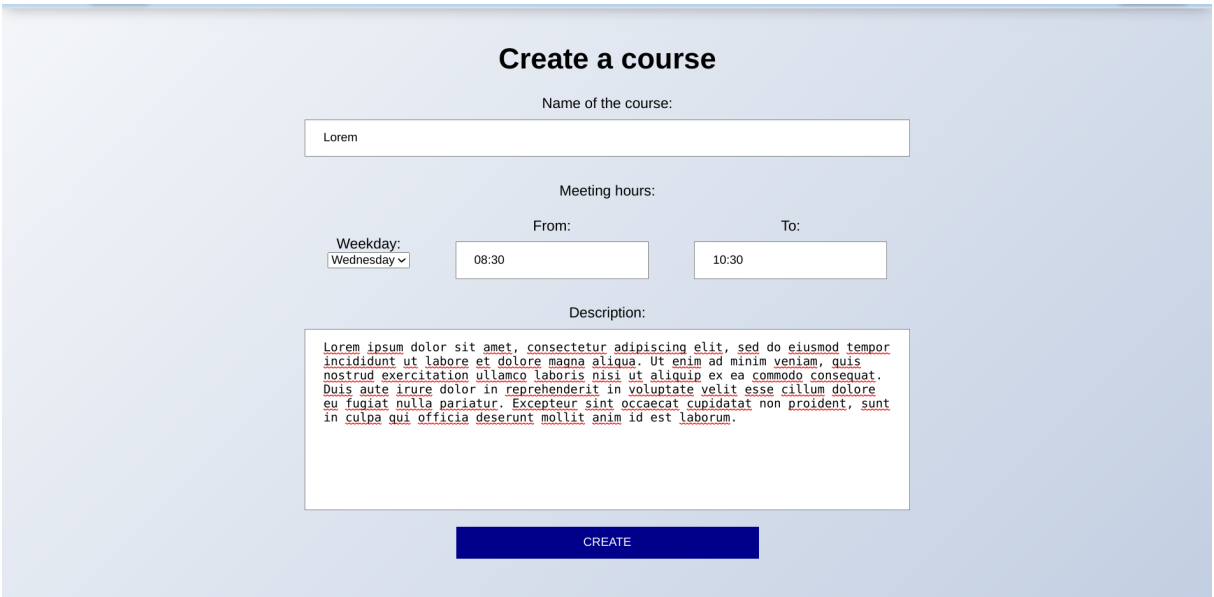

Figure 3.14: Screenshot of the course creation page, with a filled form

After the professor submits the form, the browser shows an alert pop-up with the result (success/failure) of the course creation.

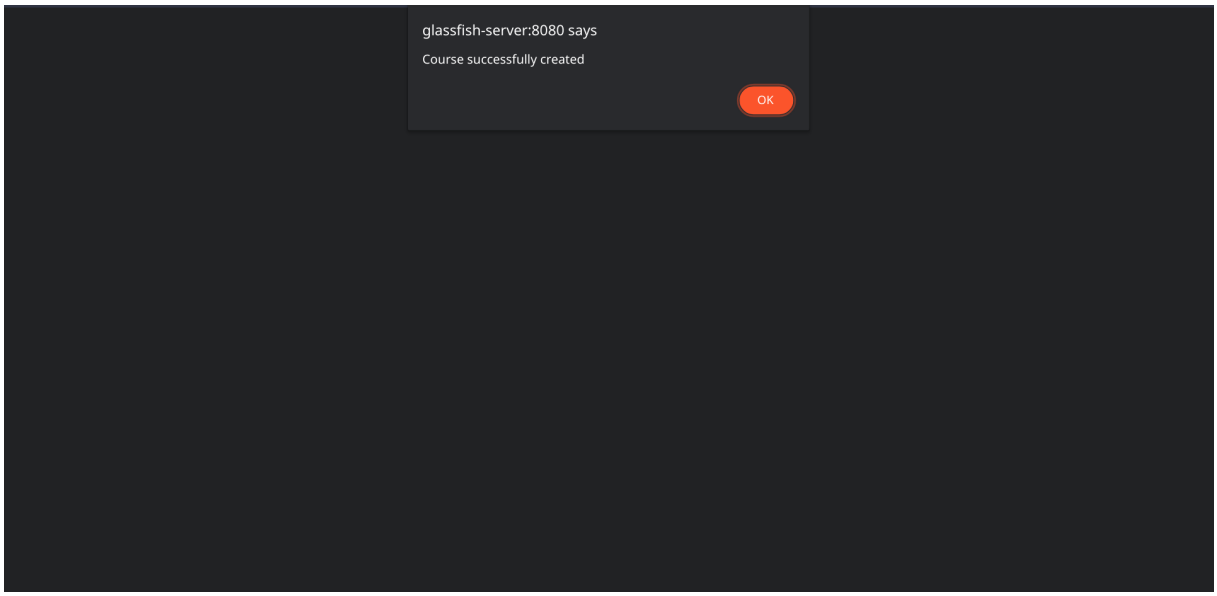

Figure 3.15: Screenshot of the course creation page, after the professor submits the form and creation succeeds

If the creation succeeds, every student will be able to view the course details page and book a meeting in the available time slots.

Figure 3.16: Screenshot of the course detail page, after the creation of the course

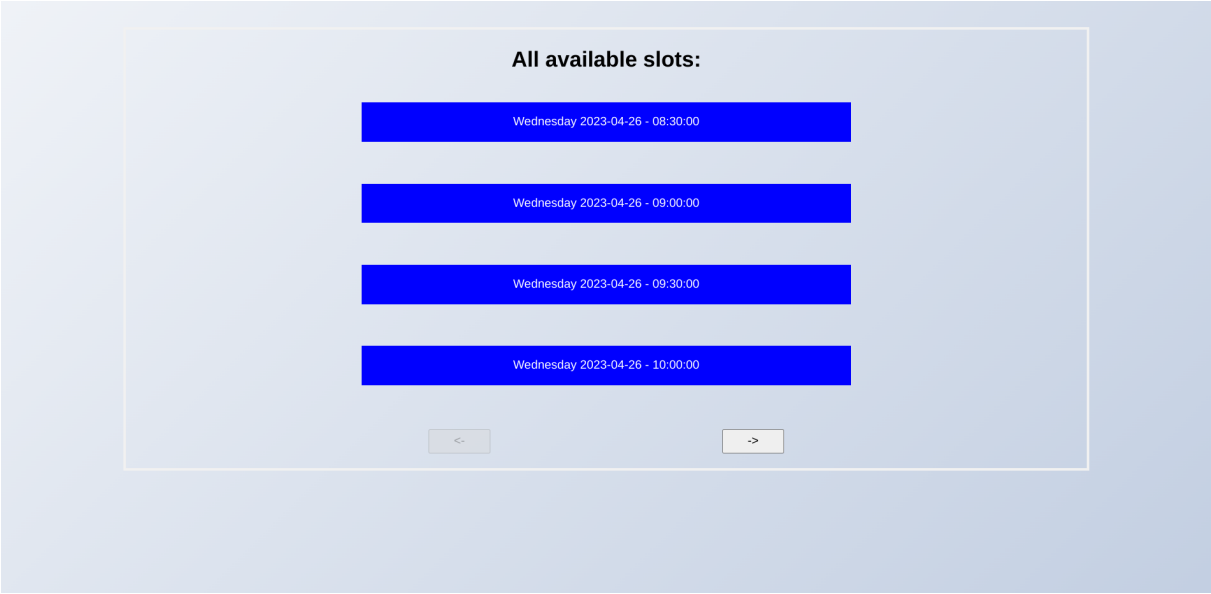

Figure 3.17: Screenshot of the page for booking a meeting, after the creation of the course

3.4.3 Show booked meetings

In this page the professor can see its scheduled meetings and cancel them.
The professor can either:

- Delete a meeting: by clicking on the corresponding button
- change month of the shown meetings: by clicking on the buttons placed under the meetings list

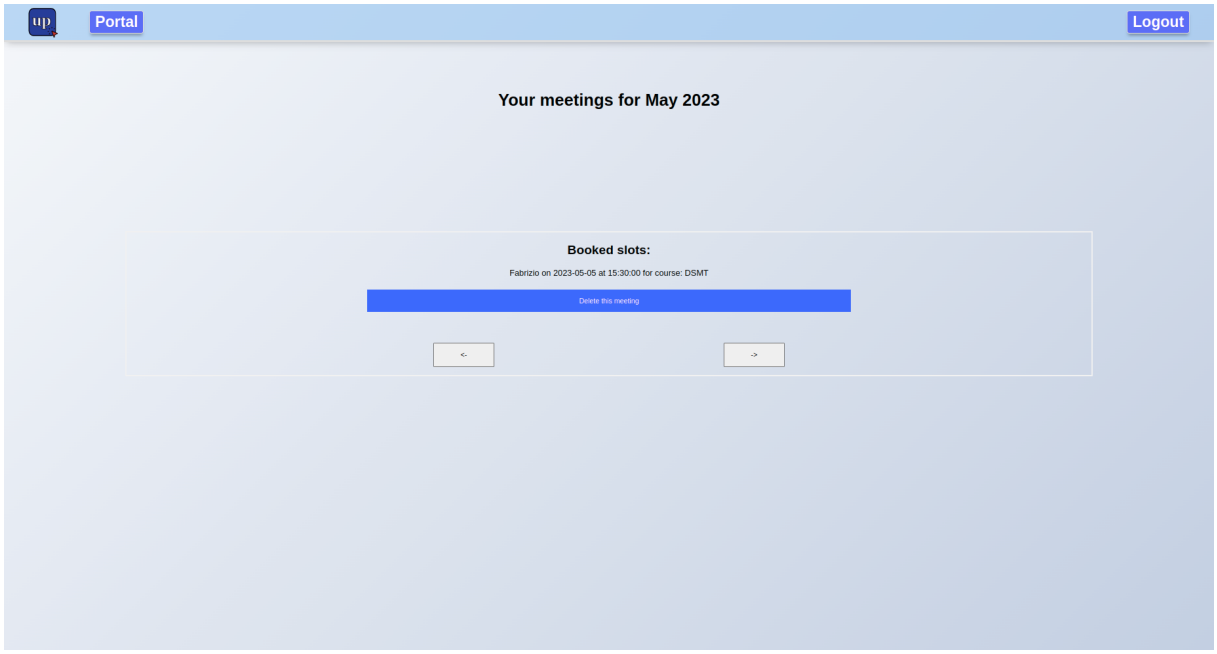

Figure 3.18: Screenshot of professor/meeting page

After the deletion of a meeting the page will show an alert with the operation result.

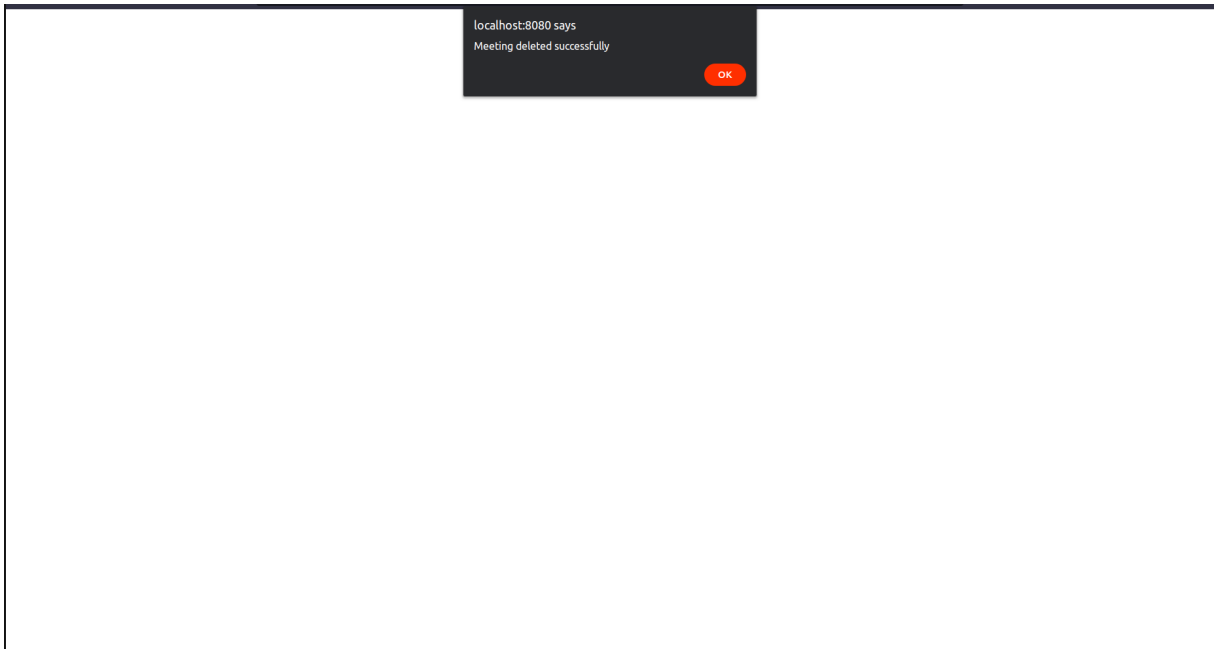

Figure 3.19: Screenshot of meeting deletion response

3.4.4 Delete a course

This is the delete course page accessible by the professor.
The professor can only delete courses that he/she has created.
To find the course to delete, he can either search for it through the search bar, or he can browse through all of his courses by hand.
Once the course is found, to delete it simply click on the related blue box.

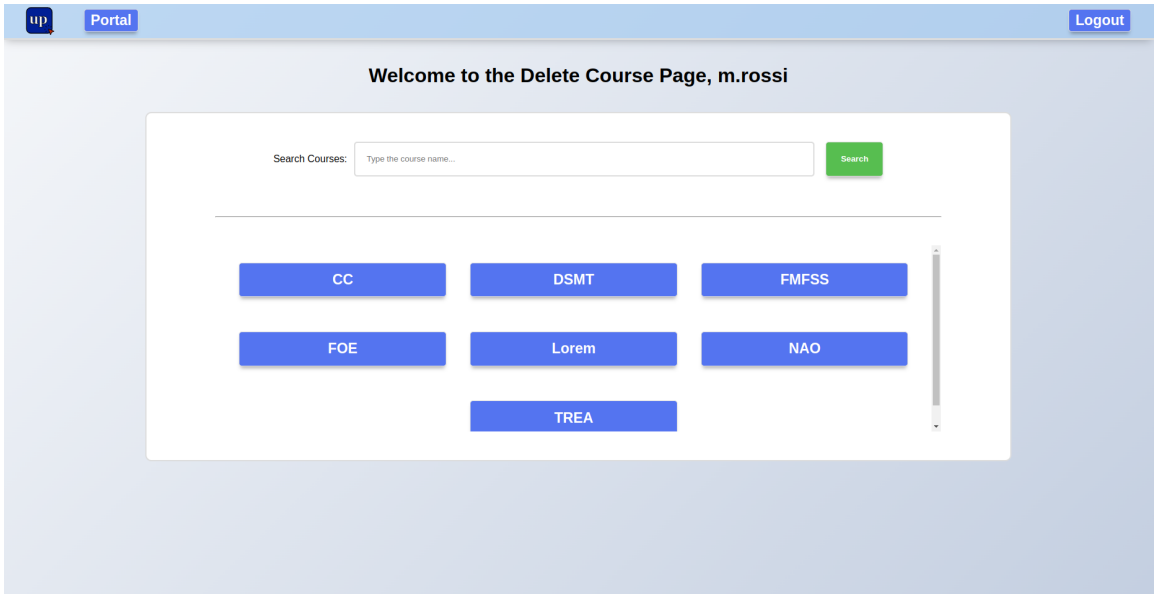

Figure 3.20: Screenshot of the delete course page.

Immediately after selecting the course to be deleted, an alert pops up and asks for the confirmation for the delete.
If the student wants to confirm he/she must click on the delete button otherwise on the x.

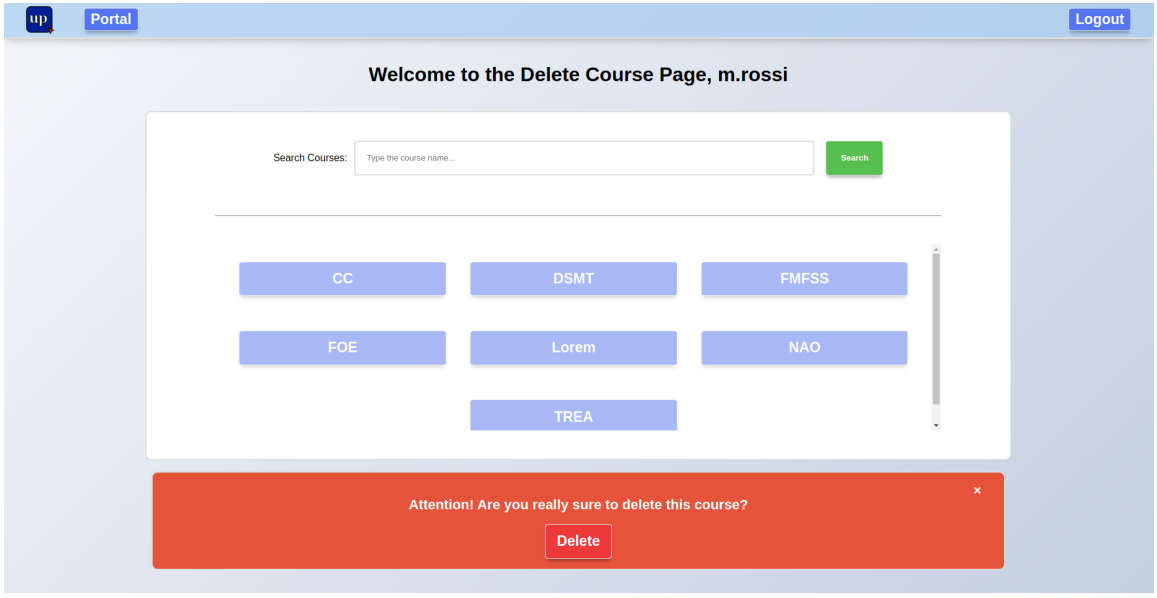

Figure 3.21: Screenshot of the delete alert of the delete course page.

The page shows a message with the result of the delete operation.

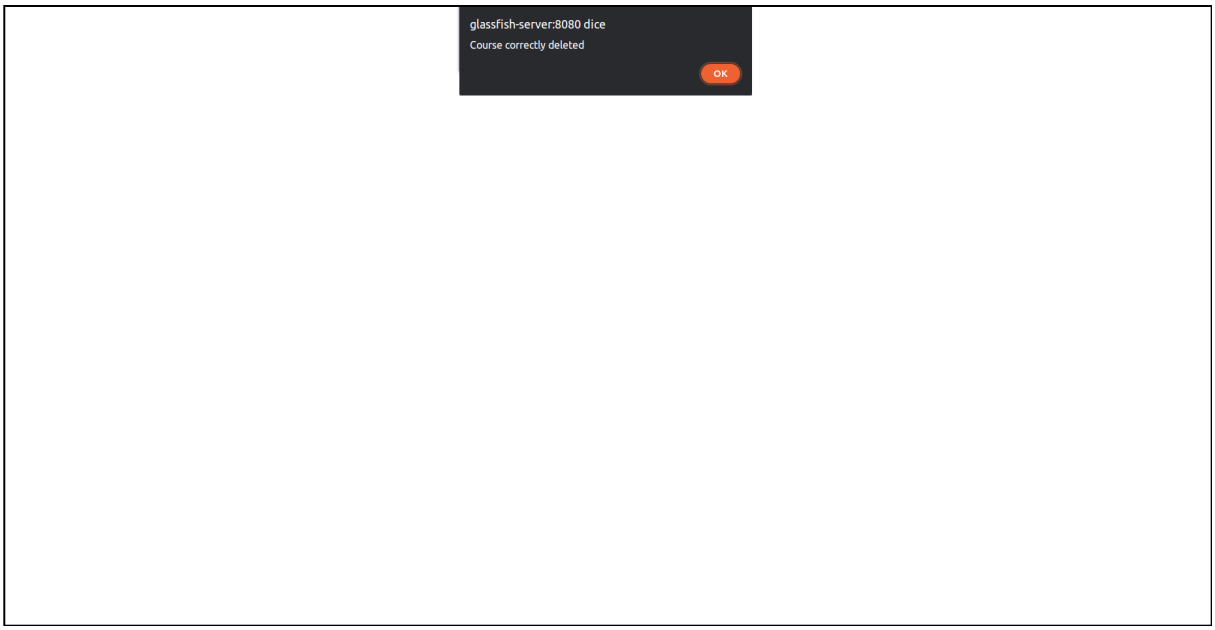

Figure 3.22: Screenshot of the delete course page after the successful delete of the course.

3.5 Admin section

3.5.1 Create professor account

On this page the admin can create a new professor's account.
To create this account, simply fill all the fields shown in the figure below and after that, click Create account.

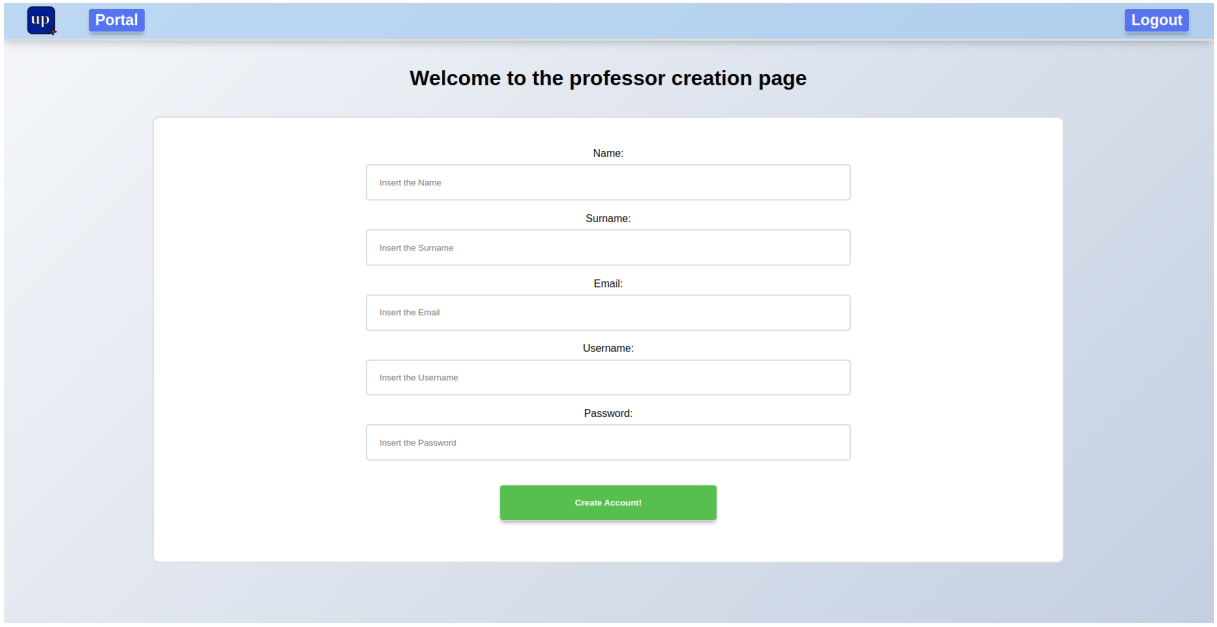

Figure 3.23: Screenshot of the create professor page.

3.5.2 User browse

In this page the admin can browse for users and ban them. The user can either:

- Browse users by username: writing in the input field positioned above the user list and by clicking the "Search" button
- Switch between the two kind of users: by clicking the button which is either "See professors" or "See students" the admin can switch the user type in which his searches are performed
- ban a user: once he found the user that he want, he can ban it clicking on the button with his name
- browse the returned list: once he searched for a specific username, he can go through the results clicking on the buttons positioned at the bottom of the page

Figure 3.24: Screenshot of user browsing page

After that the admin tried a ban operation an alert will appear to show whether the operation has been successful or not.

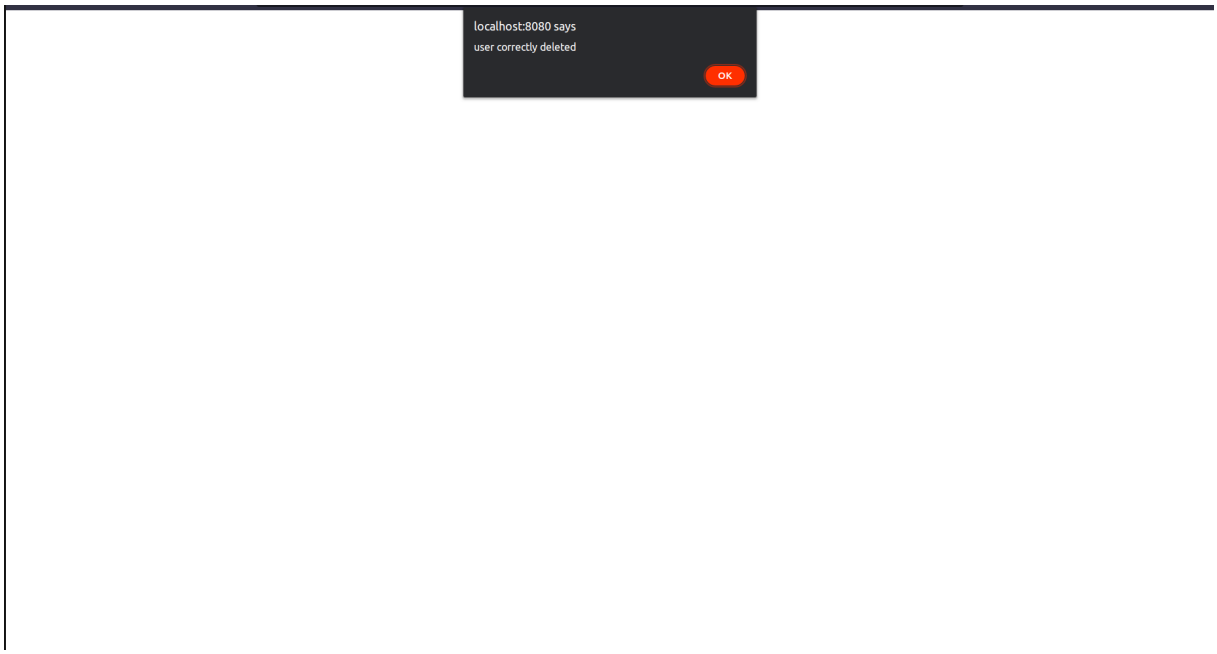

Figure 3.25: Screenshot of ban response