



EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

DEPARTMENT OF MEDIA AND EDUCATIONAL INFORMATICS

College Attendance Tracker

Supervisor:

DR. ABONYI-TÓTH ANDOR

Lecturer, PhD

Author:

Elba Kokaj

Computer Science BSc

Budapest, 2023

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

Thesis Registration Form

Student's Data:

Student's Name: Kokaj Elba

Student's Neptun code: RQWAMX

Course Data:

Student's Major: Computer Science BSc

I have an internal supervisor

Internal Supervisor's Name: Abonyi-Tóth Andor

Supervisor's Home Institution:

Department of Media and Educational

TechnologyAddress of Supervisor's Home Institution: **1117, Budapest, Pázmány**

Péter sétány 1/C. Supervisor's Position and Degree: Senior Lecturer PhD

Thesis Title: College Attendance Tracker

Topic of the Thesis:

(Upon consulting with your supervisor, give a 150-300-word-long synopsis of your planned thesis.)

Tracking attendance has always been a need during classes. It was always hard to keep track of everything, such as reasons, missed classes, attended classes, holidays, etc.

Moreover, the professors might have a hard time with every reason of the missed class considering the number of students and the tasks that they have to fulfill during the academic year.

That is why I want to create a website that is used to track attendance for universities. This website, is planned to be a help to everyone.

It will have three different roles:

- a) Professors, who will be logging on to take attendance in each session of each module,
- b) Students, who will be declaring the reason for their absence,
- c) Program Administrators who can mark attendances whenever professors cannot (e.g. took attendance in paper) and process student declarations and mark them either as absence or sickness

Administrators and Professors should also visualize attendance data in a meaningful way, both in tabular format and in charts.

Budapest, 2022. 12. 01.

ACKNOWLEDGEMENT

Firstly, I would like to extend my sincerest gratitude to my supervisor. His unyielding belief in me and unwavering guidance has been an essential cornerstone of my journey.

These past three years have been a confluence of trials and triumphs, and it is only fitting to pay tribute to those who have stood by me in my darkest and brightest moments. To my family, my unshakeable pillars of strength, thank you. Your enduring love, sacrifices and invaluable life teachings have provided me a sanctuary during the most tempestuous times.

I would also like to dedicate this work to my grandfather. The virtues, principles and lessons instilled in me by him have deeply shaped the person I am becoming. His wisdom continues to guide me and illuminate my path, and I can only hope to become half the person he is. This journey, and all its ensuing success, is a testament to his enduring influence on me.

In conclusion, it is impossible to overstate the gratitude I owe to each and every one who has played a part in this journey. This work is not just the result of my efforts, but a culmination of all your unwavering support, love, and belief in me. Each page is soaked with your faith and resilience, and for that, I am forever thankful.

Contents

1. Chapter 1	1
1.1. Purpose of the website	1
1.1. Target audience	2
1.2. Overview of main features	2
2. Chapter 2	4
2.1. Hardware requirements	4
2.2. Software requirements	4
2.3. Installation guide	4
2.4. Starting the application	6
2.5. Dependencies	7
3. Chapter 3	9
3.1. Logging in and role-based access	9
3.2. Student Dashboard	13
3.2.1. Profile	13
3.2.2. Courses	15
3.3. Professor Dashboard	17
3.3.1. Edit Profile	17
3.3.2. Classes	18
3.3.3. Taken Attendances	20
3.4. Admin Dashboard	22
3.4.1. Edit Profile	22
3.4.2. Courses	23
4. Chapter 4	26
4.1. Database schema	26
4.2. APIs and third-party integrations	27
4.3. Front-end implementation	40
4.3.1. Login Component	40
4.3.2. Student Component	41
4.3.3. Professor Component	41
4.3.4. Admin Component	42
4.3.5. UI/UX Design	42

4.3.6.	Error Handling	42
4.4.	Back-end implementation	43
4.4.1.	Database and Models	43
4.4.2.	Authentication and Authorization	47
4.4.3.	API Routes and Controllers	48
4.4.4.	Error Handling	49
4.5.	Testing	50
5.	Chapter 5	52
5.1.	Glossary of terms	52
5.1.	Reference links and resources	53

1. Chapter 1

Introduction

1.1. Purpose of the website

During my time as a university student, one of the criteria to pass a course, was to have only three missed classes. What came to my notice, after all the conversations that I had with my fellow university colleagues, is that one of our struggles was keeping track of the attended and missed classes of all the courses, considering that we had a lot of courses per semester, and a lot of work to do.

The students were not the only ones that were overloaded with work. Professors were too. Sometimes, when they tried to take attendance, they forgot if they excused a student and marked them as absent.

By the end of the course, the attendance was always a problem, because it was not tracked properly.

The reasons that are mentioned above have pushed me to come up with the Attendance Tracker, a website which helps students know how many classes they have attended and how many classes they have missed. On the other side, if the professors do any mistake while taking the attendance, the administrative staff will be able to update the taken attendance later.

So, the main purpose of the Attendance Tracker is making the university life easier and less stressful.

1.1. Target audience

The Attendance Tracker considers the students, professors and the administrative staff as the target of it.

Students are the primary group, because the passing of the course depends from the attendance. So, by using the website the students will be able to track their own attendance and monitor their activity in classes.

Moving on with the professors, it will be better if they keep track of each class of their course that is held.

Meanwhile, the administrative staff is considered as a helping hand for the professors. This is because, the professors have a lot on their plate, and if they do any mistake, the administrative staff will fix it so no student will be punished or get away with missing a class.

1.2. Overview of main features

As previously mentioned, The Attendance Tracker is a full stack website that helps track attendance.

When first opened, we see the Log In Page that contains the log in button and the forget password button. Depending on the user account we will be logged in as one of the three roles:

- a. Student
- b. Professor
- c. Administrative Staff

If logged in as a student, we will see the student side of the website. In the right hand side, we will see a text welcoming the student, and on the left hand side, we will see the student dashboard which contains:

- i. Profile view
- ii. Courses

- iii. Log out

If logged in as a professor, the website will be in the professor's role. The text on the right hand will be the same as the students, but the dashboard will be different because it will show these instead:

- i. Profile view
- ii. Classes
- iii. Taken Attendances
- iv. Log out

And lastly, if the administrative staff role is activated, the text on the right hand will remain, and the administrative staff's dashboard will have:

- i. Profile view
- ii. Classes
- iii. Log out

2. Chapter 2

System Requirements

2.1. Hardware requirements

Considering the hardware requirements, we should know that they are minimal, as all you need is a working laptop or computer that can be connected to the internet. This makes the website reachable to a big group of users, since there is no need for any additional hardware.

Furthermore, this makes it ideal for all the target groups to access the attendance data from remote locations or on the go.

2.2. Software requirements

When it comes to the software requirements, the users need to have access to any modern web browser.

The website is created using React.js for the front-end and Node.js and Express for the back-end. So, whoever tries to run the code, has to have the Node Package Manager installed on their system.

In addition, the attendance data is stored in an external MongoDB database.

2.3. Installation guide

In order to use the AttendanceTracker web application, it is essential to have a stable internet connection, along with React, Node.js, Express, and MongoDB installed in your environment.

Currently, the application is compatible with all standard web browsers. Therefore, the following instructions will guide you through the setup process for a standard web environment.

The first step towards using the application, is done using from the GitHub repository. To do this, you need to have Git installed on your computer. If Git is not installed, it should be downloaded from <https://git-scm.com/download/win>. [1]

Once Git is installed, the repository can be cloned by typing the following command in the command prompt:

```
git clone https://github.com/elbakokaj/thesis_full_stack
```

After successfully cloning the repository, you need to install the required dependencies. This process assumes you have Node.js and npm (node package manager) installed on your machine. If not, these can be installed from <https://nodejs.org/en/download/>. [2]

In the main folder of the application, you will find a file named 'package.json'. This file contains all the necessary libraries needed to run the application. You can install these dependencies by running the following command in the command prompt within the application's directory:

```
npm i
```

Afterwards, open a new terminal and direct to the client folder. In there, run:

```
npm i
```

Next, move go back to the main folder using the `cd../` command.

With this, your environment is set up, and the AttendanceTracker application is ready to be launched. Before starting the application, however, we highly recommend clearing the cache to ensure a clean run. Instructions to clear the cache depend on your specific browser, but generally, this can be done from the browser's settings or preferences.

2.4. Starting the application

After successfully installing the web application, initiating the application is the next step. To start the application, the steps below should be followed:

Open the Terminal: Access your terminal or command line interface. This could be the command Prompt on Windows, or the Terminal window in Visual Studio Code.

Navigate to the Application Directory: The web application resides in a directory on your server. Using the `cd` (change directory) command, navigate to this directory.

Start the Server: After the server software has been correctly installed and the application properly configured, you start the server using a command specific to your server software. For a Node.js server, it is typically `npm run dev`. This should be executed in the terminal within the application's directory. To start full web application you use the following command:

```
npm run dev
```

What this command does, is runs the front-end and back-end concurrently.

Upon executing the server start command, you should see output indicating that the server has started. Often, it will print a line to the effect of "Server started on port X," where X is the port number configured for the application. Also, you should see that the database has been connected successfully.

Access the Application via a Web Browser: Once the server is running, the client side will run right after, opening the default browser, followed by the port number if required. This will typically look like `http://localhost:X` for locally hosted applications.

By following the outlined steps, you have now started the web application. The application will continue to run and serve requests until the server process is stopped or the host system is shut down.

2.5. Dependencies

The dependencies are essential software packages and libraries that are used to create the website.

The back-end dependencies include:

- a. `config@3.3.9`
- b. `cors@2.8.5`
- c. `dotenv@16.0.3`
- d. `express@4.18.2`
- e. `jsonwebtoken@9.0.0`
- f. `jwt-decode@3.1.2`
- g. `jwt-encode@1.0.1`
- h. `mongoose@7.0.3`
- i. `nodemon@2.0.22`
- j. `Sequelize@6.30.0`
- k. `concurrently@8.0.1`
- l. `jest-environment-jsdom@29.5.0`
- m. `jest@29.5.0`
- n. `supertest@6.3.3`

These packages were used to handle various tasks such as configuring the application, handling cross-origin resource sharing, handling environmental variables, creating APIs, managing user authentication, testing, encoding and decoding passwords, and interacting with the database.

For the front-end, the dependencies include:

- a. @babel/plugin-transform-modules-commonjs@7.21.5
- b. @babel/preset-env@7.21.5
- c. @babel/preset-react@7.18.6
- d. @testing-library/jest-dom@5.16.5
- e. @testing-library/react@14.0.0
- f. @testing-library/user-event@14.4.3
- g. axios@1.3.5
- h. boxicons@2.1.4
- i. chart.js@4.2.1
- j. jwt-decode@3.1.2
- k. jwt-encode@1.0.1
- l. react-chartjs-2@5.2.0
- m. react-dom@18.2.0
- n. react-json-to-csv@1.2.0
- o. react-router-dom@6.9.0
- p. react-scripts@5.0.1
- q. react@18.2.0
- r. web-vitals@2.1.4
- s. msw@1.2.1
- t. identity-obj-proxy@3.0.0
- u. isomorphic-fetch@3.0.0
- v. jest-environment-jsdom@29.5.0
- w. jest@29.5.0
- x. jsonwebtoken@9.0.0

These libraries were used for tasks such as testing the application, making HTTP requests, creating charts, generating CSV files, testing, and managing the application's state.

3. Chapter 3

User Guide

3.1. Logging in and role-based access

When first entered the website, in our window the Log In page will be displayed. Depending on the email, we have three roles: Student, Professor, and Administrative Staff. The reason why we have different roles is because depending on the role we have different dashboards. What a student can do, the other two roles cannot and so on.

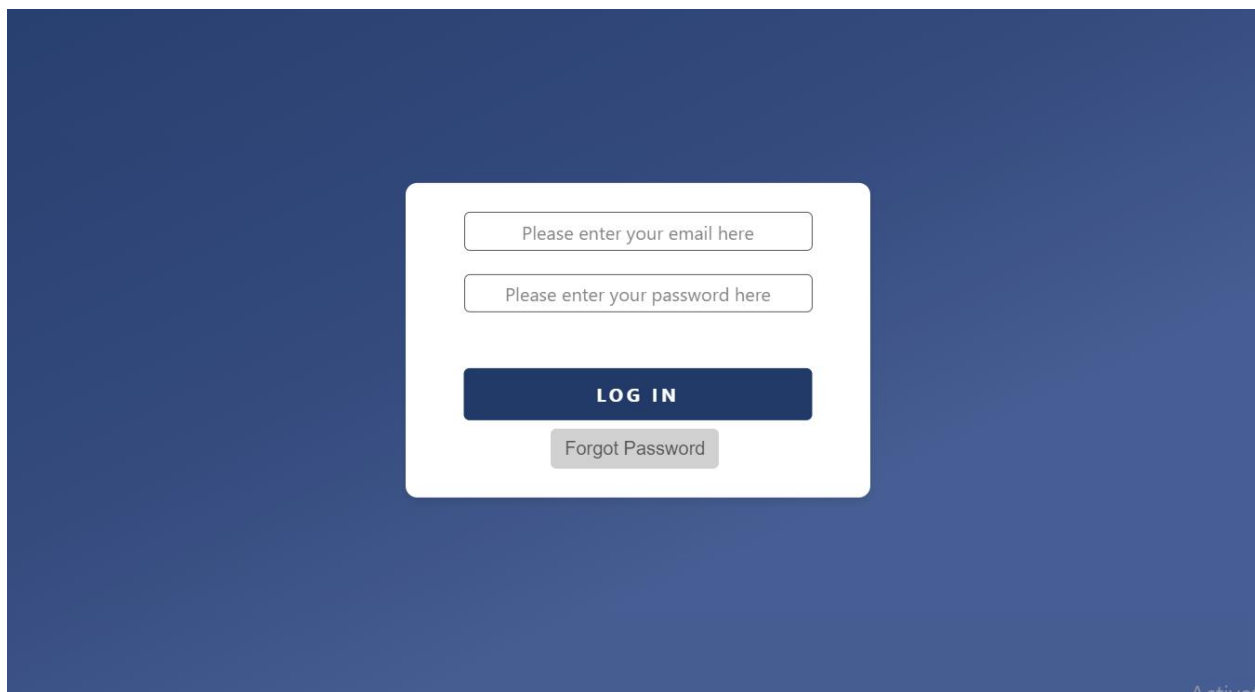


Figure 1. Log In Page

If in the email field we write something that is not an email we will get notified that we did not write an email.

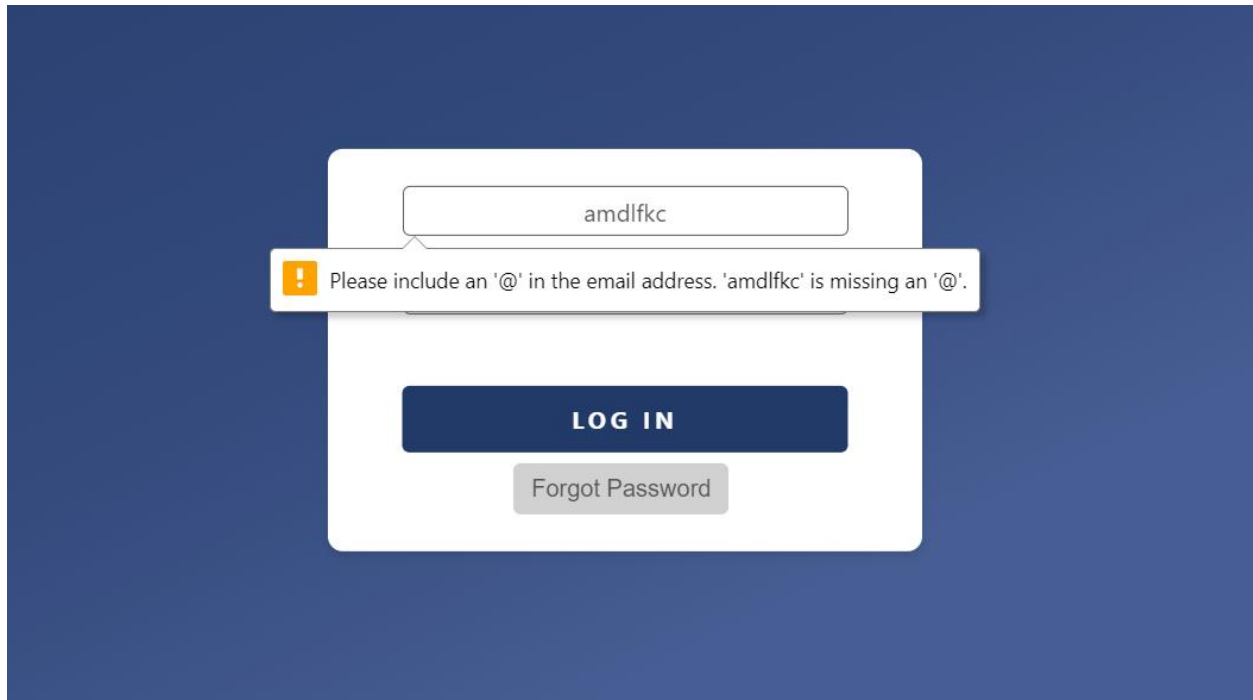


Figure 2. When the email field is not an email

One thing that may happen to a user, is forgetting the password of their account. So in the log in page we notice that we have a “Forgot Password” button.

To reset the password, we have to write the email in the email field, and afterwards click the Forgot Password button.

After doing these steps, our screen will look like following:

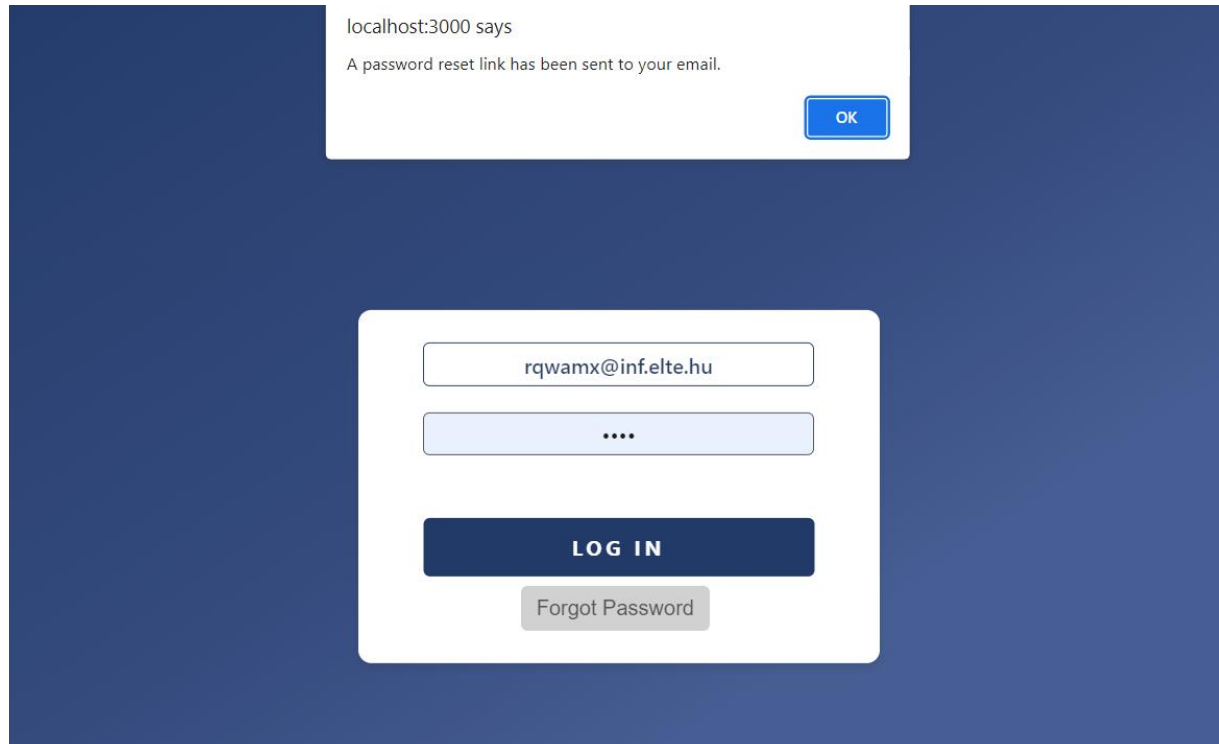


Figure 3. The shown alert when you click the Forgot Password button

In our email, we should get to reset our password.

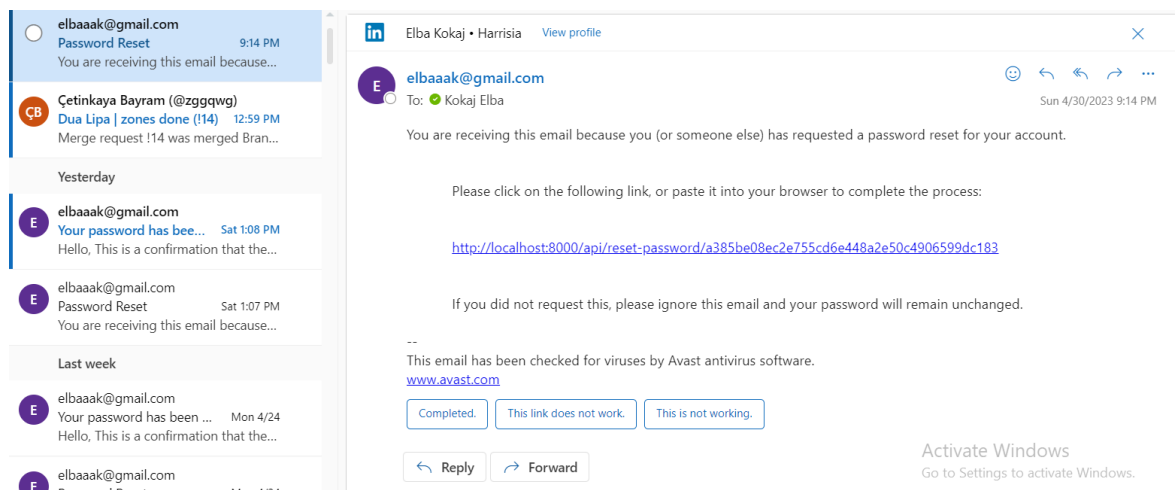


Figure 4. The first email you get after clicking Forgot Password

In the email we find a link. That link has to be clicked to send the command of changing the email. After the link is clicked, we will have a new window opened and this is what it will be displayed:

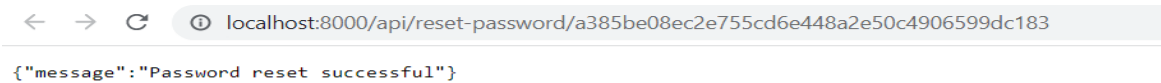


Figure 5. After clicking the link in the first email

Next, we go to the email again. In there, we will get a new email, sent from the same email address that sent us the first email, about the password change.

The next email will look like this:

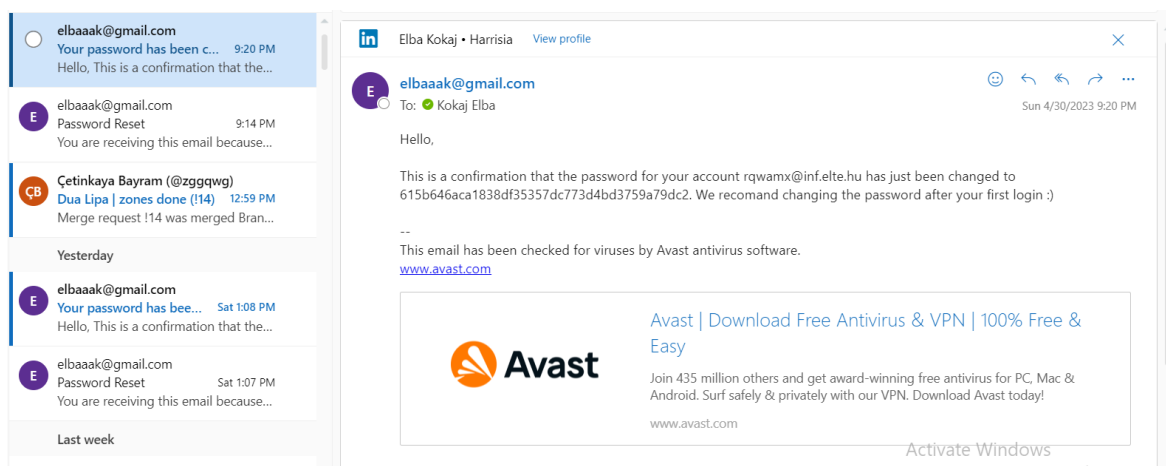


Figure 6. The second email with the new password

As noticed, we have a new password which is longer than expected. We copy this password, and go back to the log in page once again.

Now, in the password field, we place the new password that has been sent to our email.

The sender recommends changing the password, but this is up to the user. If they decide to change it, they have to navigate to the Profile section, and change the password.

The change password will be explained in the upcoming sections of the documentation.

The Log In page will also be shown when the Log Out option is clicked in either of the dashboards.

3.2. Student Dashboard

The student dashboard is shown if the user that is logging in, happens to authenticate in the student role.

What differs the student dashboard from the other two dashboards, is what is displayed in the Profile option, and what is worth mentioning is that, the Course option is a unique feature that is displayed only for the students.

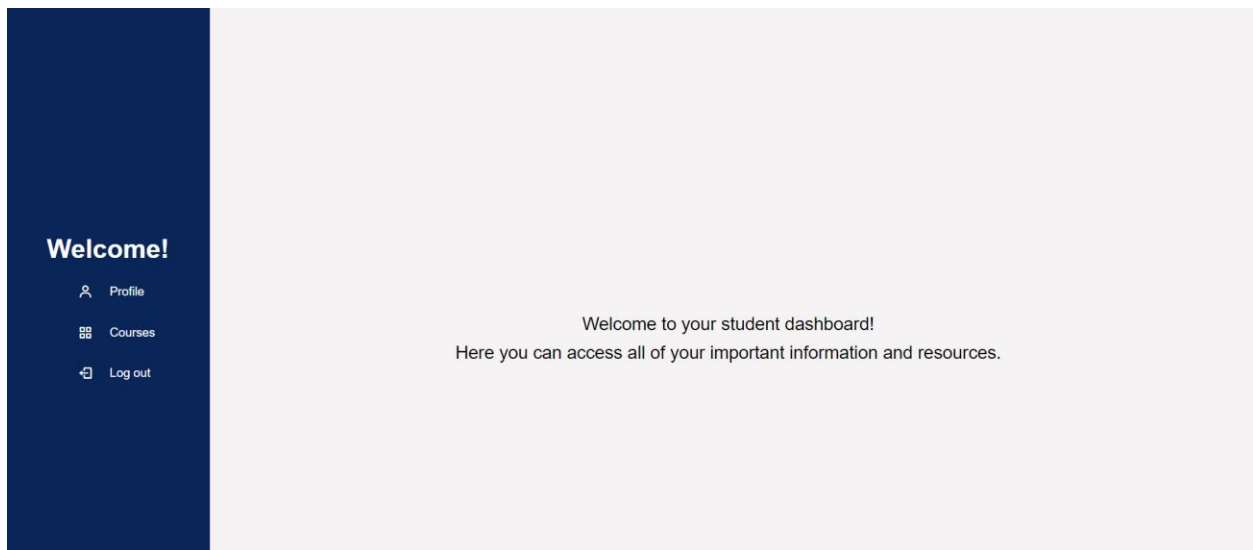


Figure 7. Student Dashboard

3.2.1. Profile

One of the elements that can be found in the student dashboard is Profile. If profile is clicked, the content on the right will be changed. It will show a white box which contains the following information of the student:

- i. Name
- ii. Surname
- iii. Birthday
- iv. Year of Enrollment

These information are followed by two buttons, the “Edit Profile” button and the “Change Password” button.

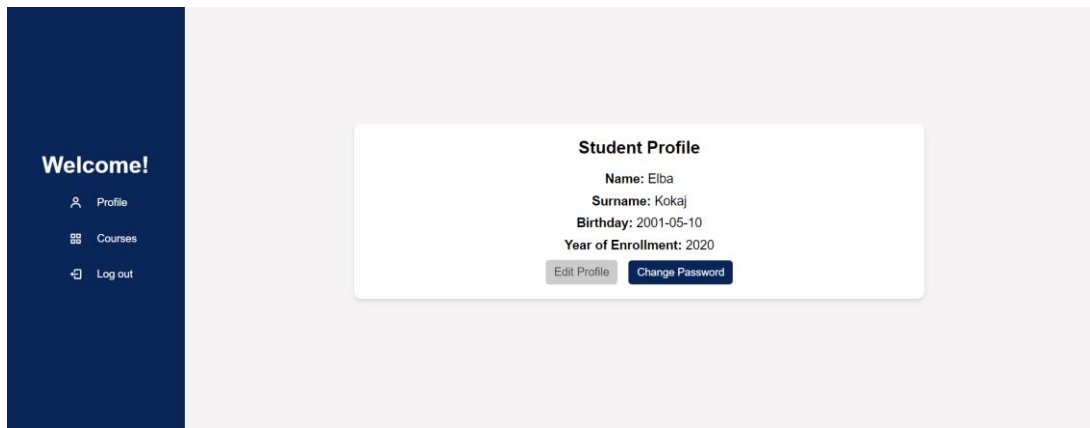


Figure 8. Student Profile

If the “Edit Profile” button is clicked the information of the student can be changed.

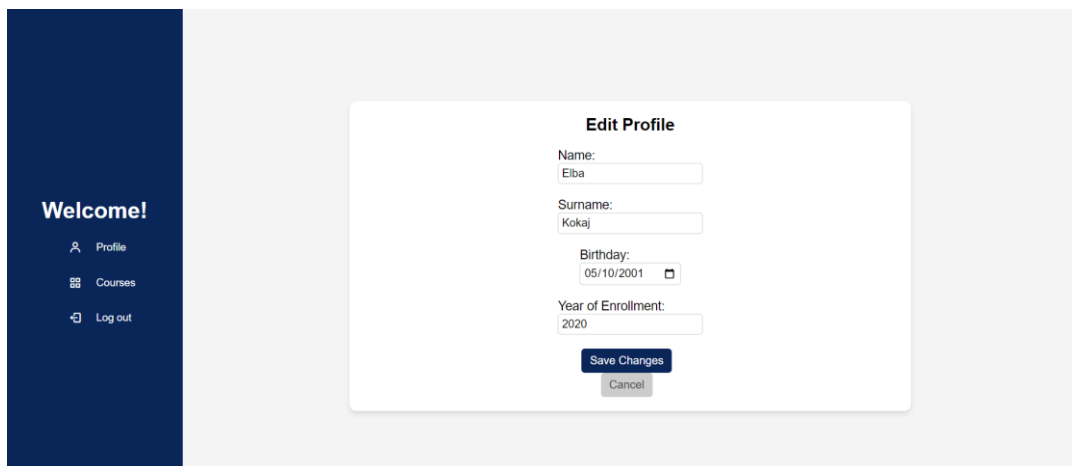


Figure 9. Editing the Student Profile

If we click the “Change Password” button, the content will show a white box containing two input fields, one being the current password and the second being the new password.

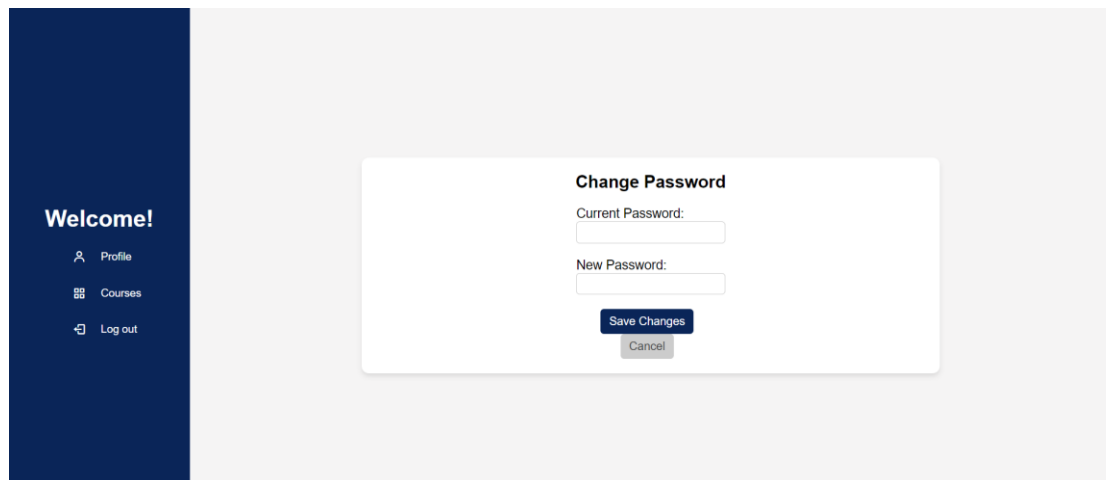


Figure 10. Changing the Student's Password

Notice how in both cases, we have the “Save Changes” button, to save what we did, and the “Cancel” button to cancel all the changes and go back to the Profile View.

3.2.2. Courses

Next up in the Student Dashboard is the Courses option. After being clicked, the content will show the list of courses that the logged in student attends. In our case, our student attends only “Introduction to Programming”, so his content will look like this:

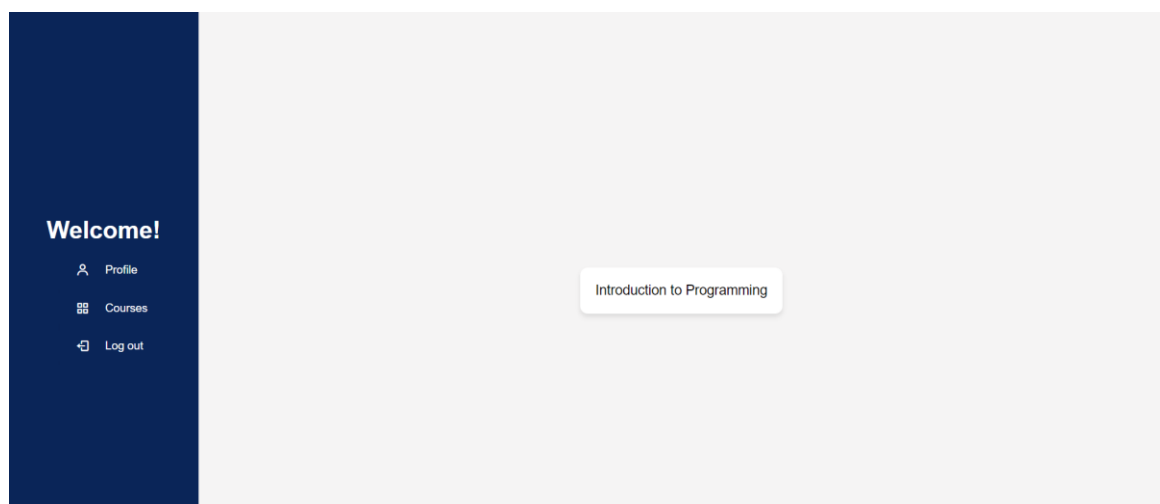


Figure 11. List of the courses the student is enrolled

Clicking the course that the student attends, will change the content. What will be shown, is what the student is interested in, so tracking his/her attendance. The attendance will be shown in a Pie Chart containing how many classes the student has missed and attended.

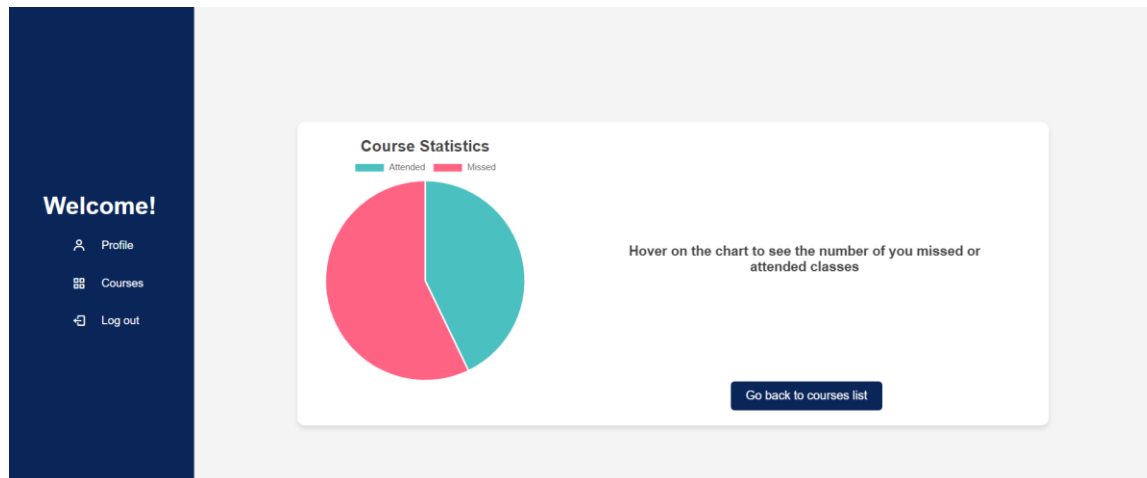


Figure 12. Course Statistics

Next to the Pie Chart, we have a text that tells us that if we hover on the chart, the number of the missed or attended classes will be seen.

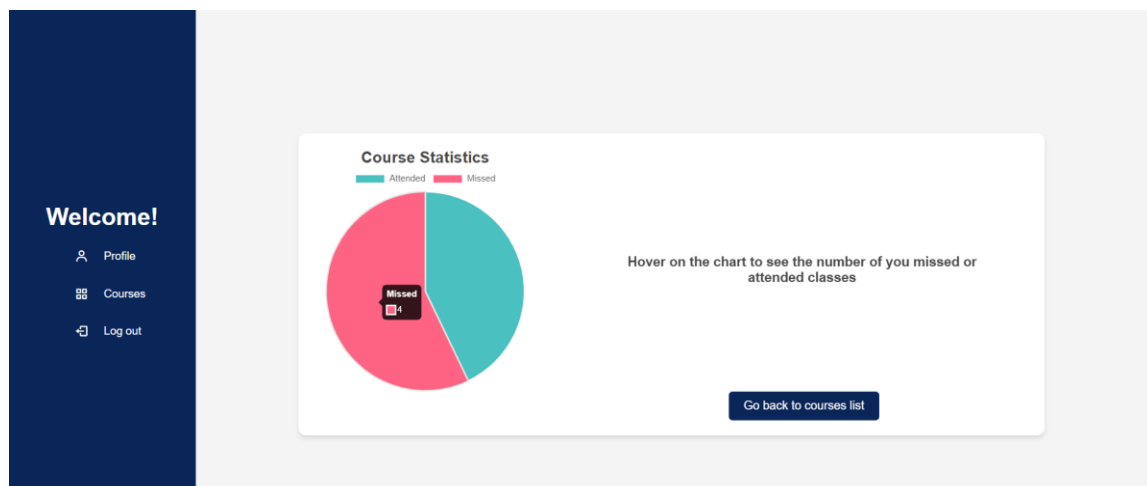


Figure 13. Getting the number of missed classes after hovering

And lastly, the “Go back to course list” button navigates us back to the course list.

3.3. Professor Dashboard

The professor dashboard is shown if the user that is logging in, happens to authenticate in the professor role.

The professor dashboard has the Profile option too but it has other information shown. Since the professor will track attendance of the students, it will have Classes in it dashboard.

What is unique about the professor dashboard, is that it has the Taken Attendances option, which cannot be found in the other two dashboards.

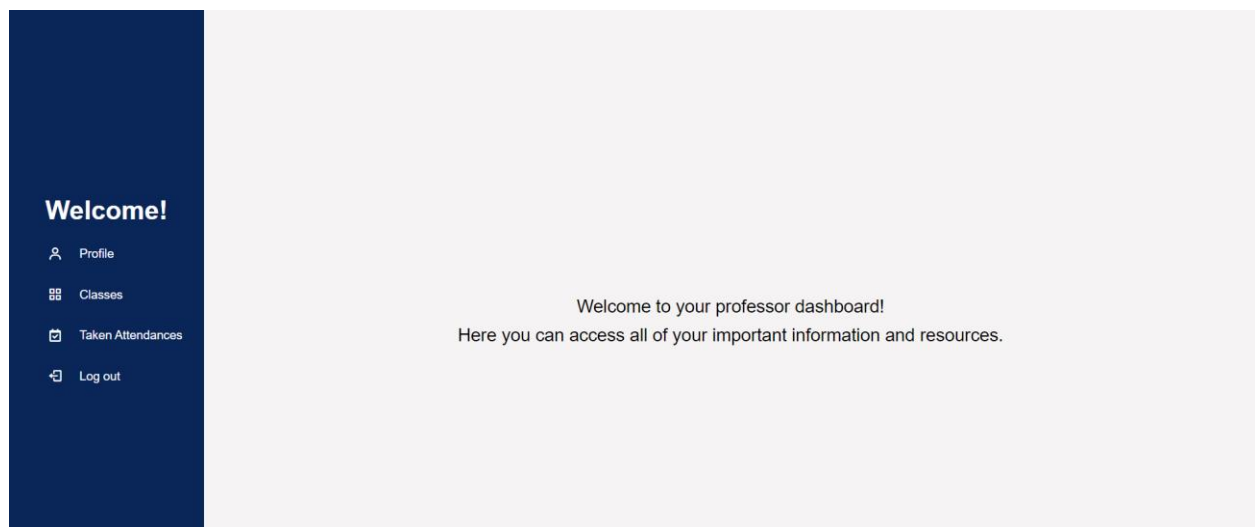


Figure 14. Professor Dashboard

3.3.1. Edit Profile

Just like in the student dashboard, the professor dashboard has the Profile in it too. But, after being clicked, the information that are shown in the Professor Profile are different from the ones in the Student Profile. The Professor information that are shown in the Profile are:

- i. Name
- ii. Surname
- iii. Birthday
- iv. Course (that he teaches)

v. Consultation hours

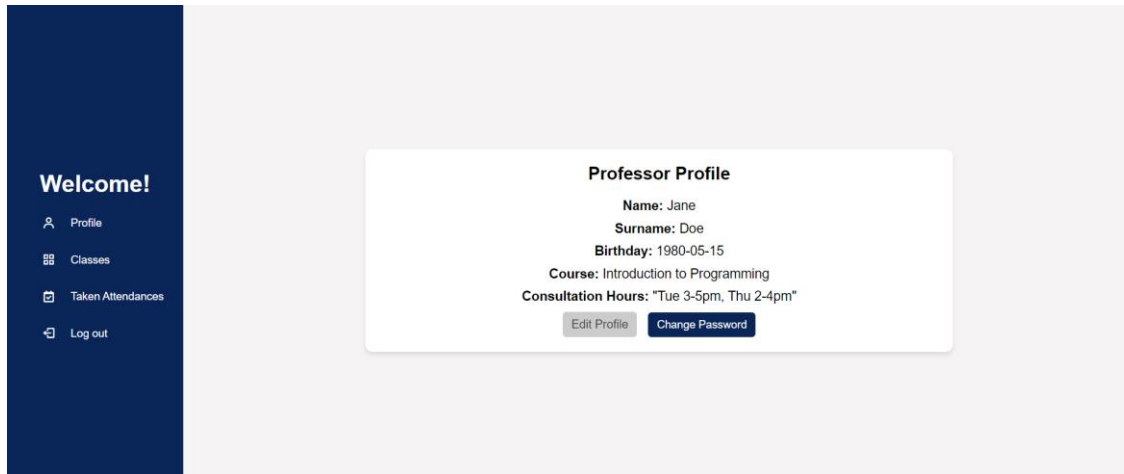


Figure 15. The profile of the Professor

As per the Edit Profile and the Change Password, they look and function the same way they do if logged in as a student.

3.3.2. Classes

To take the attendance of his class, the Professor has to click the Classes option that is found in his dashboard.

After clicking it, the student table is shown in the content, containing the name and the status buttons, which have to be clicked to decide if the student is present, absent or excused.

Welcome!

- Profile
- Classes
- Taken Attendances
- Log out

Attendance for School Year

Name	Status
Elba	<div>Present</div> <div>Excused</div> <div>Absent</div>
Alice	<div>Present</div> <div>Excused</div> <div>Absent</div>
Jay	<div>Present</div> <div>Excused</div> <div>Absent</div>

All Present

Save Attendance

Figure 16. Student table to take the attendance

Each of the buttons has a significant color, which is activated after being clicked. Present has green, excused has yellow and absent has red.

Welcome!

- Profile
- Classes
- Taken Attendances
- Log out

Attendance for School Year

Name	Status
Elba	<div>Present</div> <div>Excused</div> <div>Absent</div>
Alice	<div>Present</div> <div>Excused</div> <div>Absent</div>
Jay	<div>Present</div> <div>Excused</div> <div>Absent</div>

All Present

Save Attendance

Figure 17. Activated buttons

If all the student are present, the professor can save time by clicking the “All Present” button so he/she does not have to go one by one. In that case, if the “All Present” button is clicked, all the present buttons in the student table will be activated.

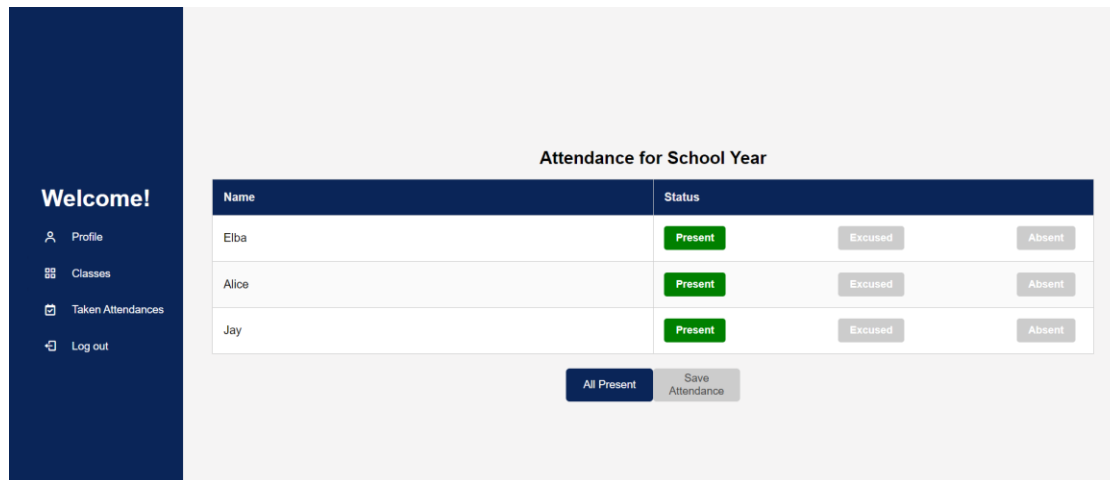


Figure 18. Each Present button activated after clicking the All Present button

And finally, when the Professor is done taking the attendance, he/she can click the “Save Attendance” button which will save the taken attendance in the database. To notify the Professor that this action is done, there will be an alert with the following message:

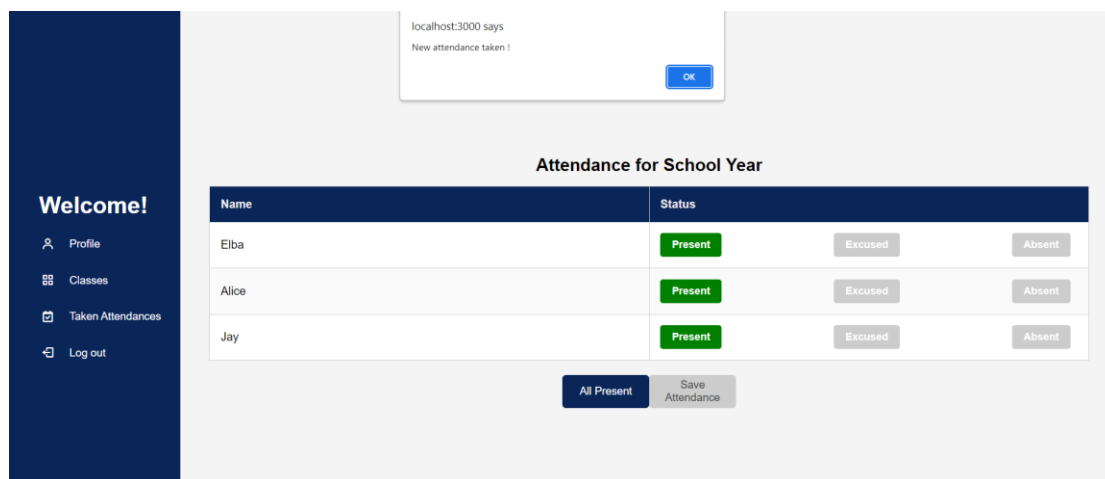


Figure 19. Alert that notifies that the new attendance has been taken

3.3.3. Taken Attendances

This feature of the Professor Dashboard, is used to get the data of the taken attendance in the date that the class was held.

When clicked, it will show all the dates of the classes that have been held.

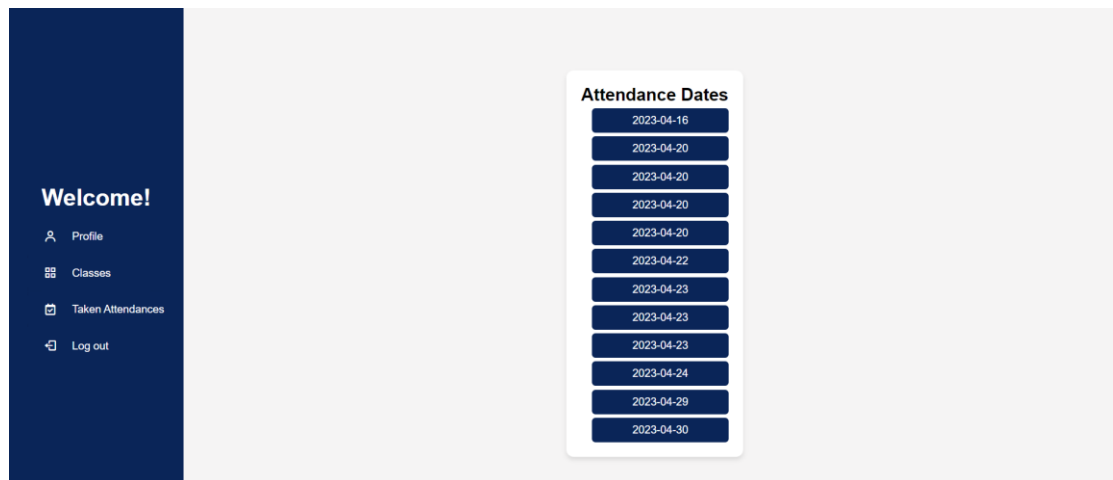


Figure 20. List of the taken attendance dates

If we click one of the dates, what will happen is, an Excel file will be downloaded. After opening the Excel file, in there we will find the name, surname and the status of the students that are saved in the database for the desired date.

So for example, in our case we decided to mark all students present. After saving it, and going to the Taken Attendances option, we decided to download that attendance list, which is 2022-04-30. After clicking the date, and the Excel file finishes downloading, we open it and see that everything is downloaded and shown properly.

124										
	A	B	C	D	E	F	G	H	I	J
1	Name	Status								
2	Elba Kokaj	present								
3	Alice Smith	present								
4	Jay Doe	present								
5										
6										
7										
8										
9										
10										
11										
12										

Figure 21. The downloaded Excel file

3.4. Admin Dashboard

After going through the authentication stage, and the user gets authenticated as an Administrative Staff user, the Admin Dashboard will be activated. What the Administrative Staff user will see is, the dashboard containing the Profile button following with the Courses.

3.4.1. Edit Profile

Just like in the other two roles, we have the Profile that will show the Profile View here to.

But, different from the other two roles, the Administrative Staff user will have the following information displayed:

- i. Name
- ii. Surname
- iii. Birthday

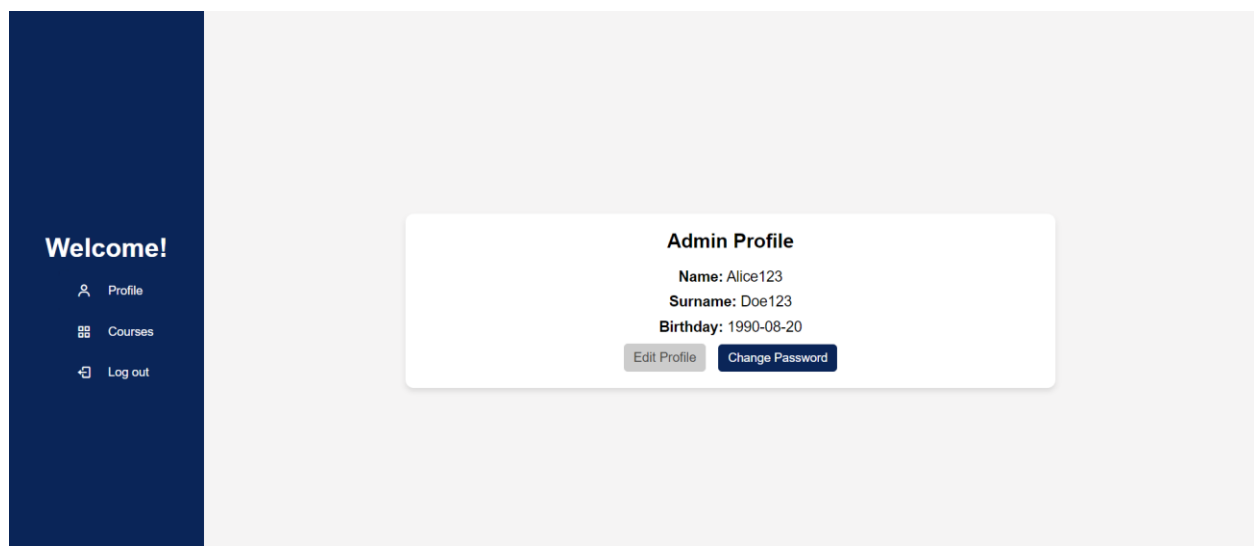


Figure 22. Administrator Profile

The “Edit Profile” and the “Change Password” buttons, are the same as for the other two roles.

3.4.2. Courses

The Courses that is found in this dashboard is different from the one that is found in the Student's Dashboard.

The difference can be seen right after clicking it, because after clicking this Courses option, we will get a list of Professors and the course that they teach.

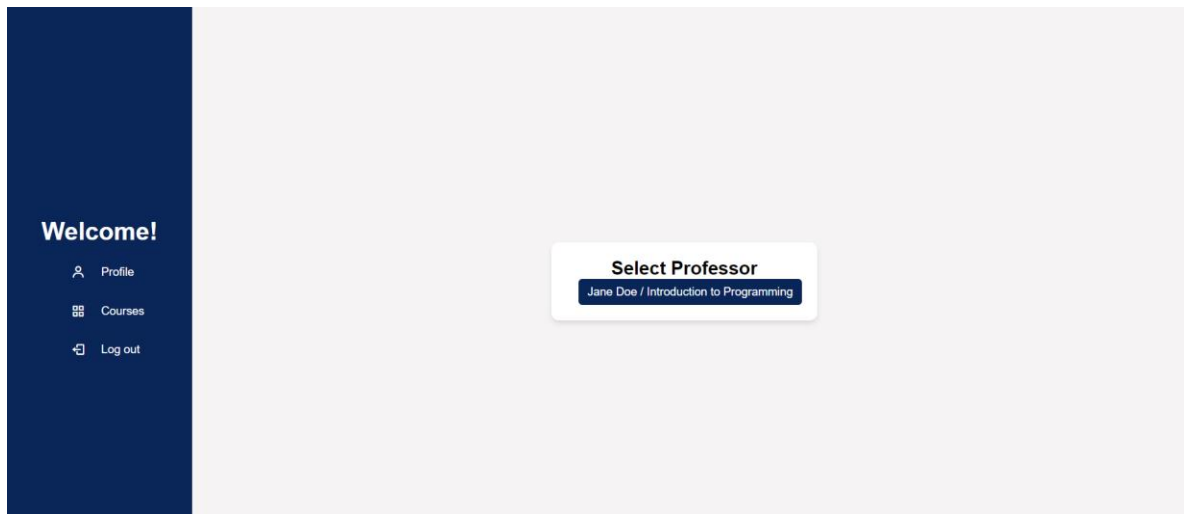


Figure 23. The list of the Professor's and the course they teach

The Administrative Staff user should choose the option that contains the information that they have to change.

After clicking on the desired Professor, all the attendances dates that the Professor saved for the course will be displayed next to the Professor Selection list.

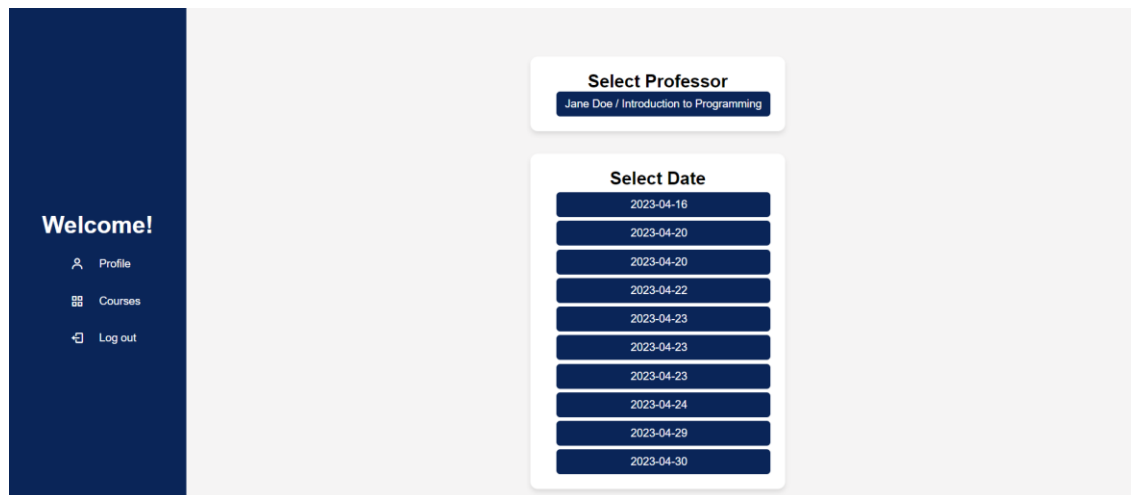


Figure 24. The dates of the taken attendances from that course

In this case, the user chooses the date of a taken attendance that has to be updated.

After choosing the date of the saved attendance, we retrieve the data from the database and display them in the same table that the Professor took it.

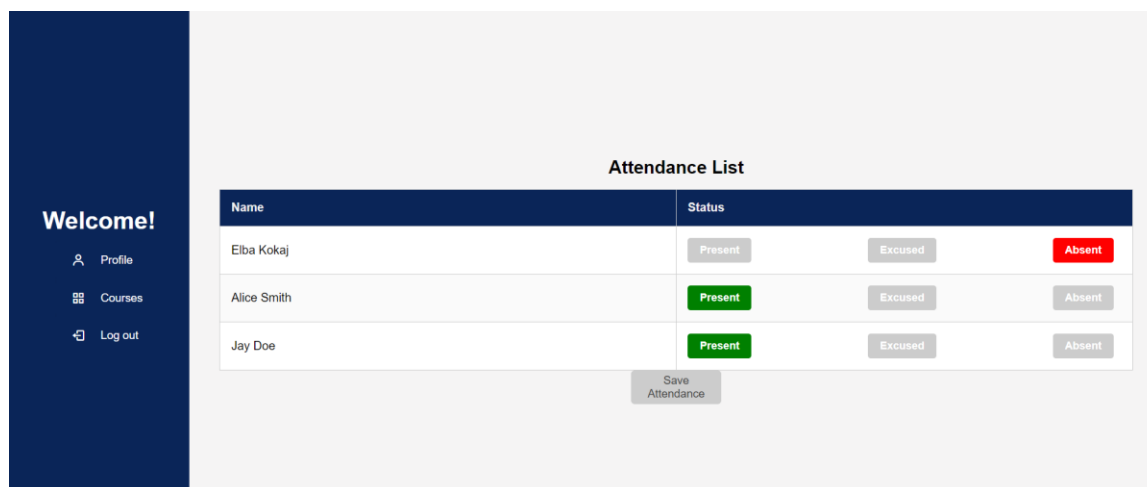


Figure 25. Elba marked as absent

So, in this case, the student called Elba is marked as absent, but in fact she was excused. The Administrative Staff user changes this by clicking the “Excused” button and after clicking the “Save Attendance” button, the attendance will be updated in the database.

Welcome!

Profile

Courses

Log out

Attendance List

Name	Status
Elba Kokaj	<div>Present</div> <div>Excused</div> <div>Absent</div>
Alice Smith	<div>Present</div> <div>Excused</div> <div>Absent</div>
Jay Doe	<div>Present</div> <div>Excused</div> <div>Absent</div>

Save Attendance

Figure 26. Elba's attendance updated to Excused

4. Chapter 4

Technical Documentation

4.1. Database schema

Using MongoDB, we have created the database that holds all the information that are shown in the web application, and that the user decides to input.

This database is an external database, which can be found in the cloud of MongoDB, after logging in with the account that has created this database.

What is worth mentioning is that, our information are stored in a clustered collection. This collection, known as our database, is called AttendanceTracker.

The AttendanceTracker database has three main tables:

i. Attendances

This table contains information such as: the id that is used to identify a taken attendance, the id of the course and the id of the group which are taken from the course table, the date when the attendance was taken and an array of records where we store the id of the student, which is taken from the users table, and the status, which can be absent, excused or present.

ii. Courses

The information that this table holds are the one that help us identify the courses that are taken in the university.

To help identify these courses and differing them from one another, we use the corresponding id of the course, the name, the professor id that we take from the user table to know who teaches that course, the course date to know when the course was created, and an array of groups. In this array, we have the group id and the group name

and another array called “studentIds”, that holds all the ids of the students that take this course.

iii. Users

All the users of our web application, can be found in this table. They differ from one another using the information that is stored. Each user has their id, the role they are logged in as (admin, professor or student), the email that is used to log in in the web application, their name and surname, their birthday, the password they have, and their authentication token.

But, what is worth mentioning, there are some fields that can be found on some users that others do not have. This is because of their role.

If they are written as a student, they will have their year of enrollment too, but if they are a professor they will have the consultation hours and the name of the course that they teach.

4.2. APIs and third-party integrations

Our web application has an API with endpoints for managing users, courses, and attendances.

Considering we have three different roles, we have different information that have to be displayed for each role, meaning we have different requests made depending on the role to manage users, courses, and attendances.

In the backend folder, there are folders depending on the roles calls: admin, professor and student. The files are called the same as the tables, but after opening them, we see the different requests that are done in there. We will go through each file of the folders to see the requests.

- i. Admin folder
 - a. Attendances file

```
router.put("/update_attendance_records", async (req, res) => {
  try {
    const { courseId, groupId, date, records } = req.body;

    console.log("req?.query:", req);
    console.log("courseId:", courseId);
    console.log("groupId:", groupId);
    console.log("date:", date);

    const attendanceRecord = await Attendances.findOne({
      courseId,
      groupId,
      date,
    });

    console.log("attendanceRecord:", attendanceRecord);

    if (!attendanceRecord) {
      return res.status(404).json({ message: "Attendance record not found" });
    }

    attendanceRecord.records = records.map((record) => ({
      studentId: record.studentId,
      status: record.status,
    }));

    const updatedAttendanceRecord = await attendanceRecord.save();

    res.status(200).json({
      message: "Attendance records updated successfully",
      attendanceRecord: updatedAttendanceRecord,
    });
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: "Internal Server Error" });
  }
});
```

Figure 27. The router that updates the attendance record

This request is a put request, which is used to update the attendance record, a function that the admin does.

First we find the needed attendance record, which can be done using the course id, group id, and the date when the attendance record was created.

If there is no such attendance, a message “Attendance record not found” will be returned.

If it is found, then we get the student id of the student that is in that record that we want to update their status, and the status of the student.

After changing it, the updated attendance record will be saved as the attendance record.

Following the update, there will also be a message that says: “Attendance records updated successfully”. The web application is prepared if an error occurs for the internal server.

b. Courses file

```
router.get("/find_specific_course/:professor_id", async (req, res) => {
  try {
    const courses = await Courses.find({ professorId: req.params.professor_id });
    const courseIds = courses.map(course => course._id);
    const attendance_dates = await Attendances.find({ courseId: { $in: courseIds } }).populate({
      path: "records.studentId",
      select: "firstName lastName",
      model: Users
    });
    console.log('courses:', courses);
    console.log('attendance_dates:', attendance_dates);
    res.json({ courses, attendance_dates });
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: "Internal Server Error" });
  }
});
```

Figure 28. The GET router to find a specific course using the professor ID

In the courses file, we find a GET router, which is used to find the specific course using the professor id. This is because we need to get the attendances that have been taken in a course.

Since when we want to update a specific attendance, in the admin dashboard, after clicking courses, we choose the professor’s name and the course he teaches, then a date of the taken attendance that we want to update. So this router gets the dates of that course that we want to edit.

c. Users file

```
router.get("/find_all_professors", async (req, res) => {
  try {
    const users = await Users.find({ "role": "professor" });
    // console.log('req', req)
    res.json(users);
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: "Internal Server Error" });
  }
});
```

Figure 29. Finding all the users in the role of Professor

This router is the one that we use in the toggle courses function in the end. This helps us get all the users that are in the role of the professor.

d. Profile file

```
router.put('/edit/:admin_id', async (req, res) => {
  const id = req.params.admin_id;
  const updatedData = {};

  if (req.body.firstName) {
    updatedData.firstName = String(req.body.firstName);
  }
  if (req.body.lastName) {
    updatedData.lastName = String(req.body.lastName);
  }

  if (req.body.birthday) {
    updatedData.birthday = String(req.body.birthday);
  }

  const options = { new: true };

  try {
    const updatedProduct = await Users.findByIdAndUpdate(id, updatedData, options);

    res.json(updatedProduct);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Server error' });
  }
});
```

Figure 30. The PUT router that edits the administrator's information

This is the router that helps the admin edit their profile. The PUT indicates that the user is putting information in the database. Identifying the user as the admin using the id is what happens first, and then moving on to checking the which one of the admin profile information is updated, to put it in the database as the new one.

```
router.get("/:admin_id", async (req, res) => {
  try {
    const users = await Users.find({ "_id": req.params.admin_id });
    res.json(users);
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: "Internal Server Error" });
  }
});
```

Figure 31. GET router to know when a user is logged in the administrator role

This router is used to know that the application has to show the admin dashboard.

```
router.put('/change_password/:admin_id', async (req, res) => {
  const id = req.params.admin_id;
  console.log('req.body', req.body)
  try {
    const foundUser = await Users.findByIdAndUpdate(id, {});
    const oldPassEnc = sign(req.body.old_password, secret)
    const oldPassMatch = oldPassEnc == foundUser?.password;
    console.log('oldPassMatch', oldPassEnc, foundUser?.password)

    if (oldPassMatch == true) {
      const updatedData = {};
      if (req.body.new_password) {
        const encPass = sign(req.body.new_password.toString(), secret)
        updatedData.password = encPass;
      }
      const options = { new: true };
      await Users.findByIdAndUpdate(id, updatedData, options);
      res.json({ message: "Password changed sucesfully!" });
    }
    else {
      res.json({ message: "Old passwords do not match!" });
    }
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Server error' });
  }
});
```

Figure 32. Changing the password of the administrator

Just as the Edit Profile router, the Change Password is a PUT router too.

When the admin decides to change the password, first he/she has to input the old password and then the new password. The password is changed only if the old password that the admin wrote in the change password section matches the one in the database.

- ii. Professor folder
 - a. Attendances file

```
router.get("/find_students/:course_id", async (req, res) => {
  try {
    const allAttendances = await Attendances.findOne({ courseId: req.params.course_id });
    const allUsers = await Users.find();

    const records = allAttendances?.records?.filter((el) => el.studentId);
    // console.log('records', records)
    const all_students_that_attend = [];
    for (const element of records) {
      const student = allUsers.find((user) => user._id.toString() === element.studentId.toString());
      if (student) {
        var name = student?.firstName
        var status = element
        const myJSON = {
          name, status: {
            studentId: element.studentId,
            status: "",
            _id: element._id
          },
        },
        all_students_that_attend.push(myJSON);
      }
    }
    res.json(all_students_that_attend);
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: "Internal Server Error" });
  }
});
```

Figure 33. Router that gets all the student that attend the course a professor teaches

This router is used to find all the students that attend the course that the professor who is logged in as, teaches.

This is used to show the students in the attendance table in the frontend.

```

router.get("/find_taken_attendances", async (req, res) => {
  try {
    const attendances = await Attendances.find().populate({
      path: "records.studentId",
      select: "firstName lastName",
      model: Users
    });
    res.json(attendances);
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: "Internal Server Error" });
  }
});

```

Figure 34. The route that helps us GET all the taken attendances

This router is used to get all the attendances that have been taken from this professor, and show them in the taken attendances section in the frontend.

```

router.post("/store_students_attendances", async (req, res) => {
  try {
    const { courseId, groupId, date, attendanceRecords } = req.body;

    // console.log('attendanceRecords', attendanceRecords)
    const newRecords = await attendanceRecords.map((attendance) => ({
      studentId: attendance?.status?.studentId,
      status: attendance?.status?.status,
    }));
    // console.log('courseId', courseId)
    const attendanceRecord = await Attendances.create({
      _id: new mongoose.Types.ObjectId().toString(),
      courseId,
      groupId: new mongoose.Types.ObjectId().toString(),
      date: new Date(),
      records: newRecords,
    });
    res.status(200).json({ message: "Attendance saved successfully", attendanceRecord });
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: "Internal Server Error" });
  }
});

```

Figure 35. The used route to save the attendance in the database

This is the router that posts the attendance that is being taken in the database.

b. Profile file

```
router.get("/:professor_id", async (req, res) => {
  try {
    const users = await Users.find({ "_id": req.params.professor_id });
    res.json(users);
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: "Internal Server Error" });
  }
});
```

Figure 36. The route that shows the Professor Dashboard

This router is used to know that the application has to show the professor dashboard.

```
router.put('/edit/:professor_id', async (req, res) => {
  const id = req.params.professor_id;
  const updatedData = {};

  if (req.body.firstName) {
    updatedData.firstName = String(req.body.firstName);
  }
  if (req.body.lastName) {
    updatedData.lastName = String(req.body.lastName);
  }
  if (req.body.course) {
    updatedData.course = String(req.body.course);
  }
  if (req.body.birthday) {
    updatedData.birthday = String(req.body.birthday);
  }
  if (req.body.consultationHours) {
    updatedData.consultationHours = String(req.body.consultationHours);
  }
  const options = { new: true };

  try {
    const updatedProduct = await Users.findByIdAndUpdate(id, updatedData, options);

    res.json(updatedProduct);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Server error' });
  }
});
```

Figure 37. The route to update the Professor's information

This is the router that helps the professor edit their profile. The PUT indicates that the user is putting information in the database. Identifying the user as the professor using the id is what happens first, and then moving on to checking which one of the professor profile information is updated, to put it in the database as the new one.

```
router.put('/change_password/:professor_id', async (req, res) => {
  const id = req.params.professor_id;
  console.log('req.body', req.body)
  try {
    const foundUser = await Users.findByIdAndUpdate(id, {});
    const oldPassEnc = sign(req.body.old_password, secret)
    const oldPassMatch = oldPassEnc == foundUser?.password;
    console.log('oldPassMatch', oldPassEnc, foundUser?.password)

    if (oldPassMatch == true) {
      const updatedData = {};
      if (req.body.new_password) {
        const encPass = sign(req.body.new_password.toString(), secret)
        updatedData.password = encPass;
      }
      const options = { new: true };
      await Users.findByIdAndUpdate(id, updatedData, options);
      res.json({ message: "Password changed sucesfully!" });
    }
    else {
      res.json({ message: "Old passwords do not match!" });
    }
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Server error' });
  }
});
```

Figure 38. The route that changes the password

Just as the Edit Profile router, the Change Password is a PUT router too.

When the professor decides to change the password, first he/she has to input the old password and then the new password. The password is changed only if the old password that the professor wrote in the change password section matches the one in the database.

- iii. Student folder
 - a. Attendances file

```
router.get("/find_specific_status", async (req, res) => {
  try {
    // Get the attendance data
    const attendanceData = await Attendances.find();
    const records = attendanceData.map((el) => el.records);
    const result = records.reduce((acc, val) => acc.concat(val), []) // flatten the records array
      .filter((record) => record?.studentId == req.query.student_id);
    console.log("result", result);

    var present = 0
    var absent = 0
    for (let index = 0; index < result.length; index++) {
      if (result[index]?.status == "present") {
        present++;
      }
      if (result[index]?.status == "absent") {
        absent++;
      }
    }
    const counts = { present: present, absent: absent }
    res.json(counts);
  } catch (err) {
    console.error(err);
    res.status(500).send('Server error');
  }
});
```

Figure 39. The route that gets the present and absent status

This router is used to get the specific status that the student needs.

Since there is a pie chart that shows how many classes the student has missed or attended, this router helps us find these so they are shown in the pie chart.

b. Courses file

```
async function getCourses(studentId) {
  try {
    const courses = await Courses.aggregate([
      { $match: { "group.studentIds": studentId } },
      {
        $project: {
          _id: 0,
          name: 1,
          semester: 1,
          professorId: 1,
          course_date: 1
        }
      }
    ]);

    if (courses.length > 0) {
      console.log(courses);
      return courses;
    } else {
      return null;
    }
  } catch (err) {
    throw err; // propagate the error up
  }
}

router.get('/:course_name/:studentId', async (req, res) => {
  const studentId = req.params.studentId;
  try {
    const courses = await getCourses(studentId);

    if (courses) {
      res.send(courses);
    } else {
      res.status(401).send('No courses found this student!');
    }
  } catch (err) {
    console.error(err); // Log error for debugging
    res.status(500).send({ error: "Internal Server Error" });
  }
});
```

Figure 40. The route that shows the courses the student is enrolled

In the frontend, after clicking the Courses button, the courses are shown. They are shown thanks to this router that exists in our APIs.

c. Profile file

```
router.get("/:student_id", async (req, res) => {
  try {
    const users = await Users.find({ "_id": req.params.student_id });
    res.json(users);
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: "Internal Server Error" });
  }
});
```

Figure 41. The route that shows the Student Dashboard

This router is used to know that the application has to show the student dashboard.

```
router.put('/edit/:student_id', async (req, res) => {
  const id = req.params.student_id;
  const updatedData = {};

  if (req.body.firstName) {
    updatedData.firstName = String(req.body.firstName);
  }
  if (req.body.lastName) {
    updatedData.lastName = String(req.body.lastName);
  }
  if (req.body.yearOfEnrollment) {
    updatedData.yearOfEnrollment = String(req.body.yearOfEnrollment);
  }
  if (req.body.birthday) {
    updatedData.birthday = String(req.body.birthday);
  }

  const options = { new: true };

  try {
    const updatedProduct = await Users.findByIdAndUpdate(id, updatedData, options);
    res.json(updatedProduct);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Server error' });
  }
});
```

Figure 42. The route that edits the student's information

This is the router that helps the student edit their profile. The PUT indicates that the user is putting information in the database. Identifying the user as the student using the id is what happens first, and then moving on to checking which one of the student profile information is updated, to put it in the database as the new one.

```
router.put('/change_password/:student_id', async (req, res) => {
  const id = req.params.student_id;
  console.log('req.body', req.body)
  try {
    const foundUser = await Users.findByIdAndUpdate(id, {});
    const oldPassEnc = sign(req.body.old_password, secret)
    const oldPassMatch = oldPassEnc === foundUser?.password;
    console.log('oldPassMatch', oldPassEnc, foundUser?.password)

    if (oldPassMatch == true) {
      const updatedData = {};
      if (req.body.new_password) {
        const encPass = sign(req.body.new_password.toString(), secret)
        updatedData.password = encPass;
      }
      const options = { new: true };
      await Users.findByIdAndUpdate(id, updatedData, options);
      res.json({ message: "Password changed sucesfully!" });
    }
    else {
      res.json({ message: "Old passwords do not match!" });
    }
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Server error' });
  }
});
```

Figure 43. The route that changes the password of the student

Just as the Edit Profile router, the Change Password is a PUT router too.

When the student decides to change the password, first he/she has to input the old password and then the new password. The password is changed only if the old password that the student wrote in the change password section matches the one in the database.

4.3. Front-end implementation

This chapter is about discussing the design that is used and the implementation of the components that are in the front-end of our web application.

For the client-side, we have React and Redux as our primary technologies, along with Axios that is used to handle the HTTPS requests to our back-end API.

As previously mentioned, this website can be used as an educational portal where students check their attendance status that is managed from the professors of their courses. Not only are these the roles that have impact in our web application, but there are administration functions found too for managing the attendance.

Considering these information, this chapter will focus on the components of the website, which are:

- The Login Component
- The Student Component
- The Professor Component
- The Admin Component

4.3.1. Login Component

Any user that first enters the application, will have this component as the entry point.

The Login Component is the component that is responsible for authenticating the user and directing them to the appropriate dashboard, based on their user role.

The user starts by providing the email and their password. After the submission, the component sends a request to our back-end API for authentication. If this succeeds, the user will have their corresponding data stored in the local storage, being redirected to their appropriate dashboard.

If the authentication is failed, then the user is shown an error message.

4.3.2. Student Component

The Student component is the user interface that serves the users that are logged in as students.

Here, the student user is able to view their attendance record, which includes details of how many classes they have missed and attended for each course that they take. This attendance data is fetched from the back-end API whenever this component is loaded.

Besides checking their attendance data, this component provides the ability to view their profile, including making changes to it, such as updating their name, surname, email and password. If these changes are made, they are sent to the back-end API to be stored in the database.

4.3.3. Professor Component

The main interface for professors, is the Professor component. Here, professors can view their profile information, and also, make changes. These changes are stored in the database if a request is sent to the back-end API.

What the professors can do, is to view and manage the attendance of their students.

When it comes to managing the attendance, the professor is shown a list of their students, each of which can be marked as present, excused, or absent. If all the students are present, the professor has the option to mark all students as present, and to save the attendance record for the day. The attendance record are sent to the back-end API for storage.

Also, Professors can view a list of dates for which attendance has been taken, and for a selected date, they can download the taken attendance record.

4.3.4. Admin Component

From this component, the administrators have the ability to edit the taken attendances, including their profile information that can be found in the profile section of the administrator user.

When the administrator want to edit a taken attendance, the component fetches attendance data from the back-end API, displays it in a table, and provides the ability to edit it.

Changes are sent to the back-end API and later on stored in the database.

4.3.5. UI/UX Design

Our design philosophy for this project was to keep things simple, intuitive, and easy to navigate. The interface is designed with clear labels and instructions, along with visual cues such as color coding for attendance status. We also prioritized responsiveness, so that the site can be easily used on both desktop and mobile devices.

4.3.6. Error Handling

In our front-end components, we incorporated robust error handling and user input validation. This includes checking for required fields, and providing clear error messages when something goes wrong.

Error conditions such as unsuccessful API requests are handled gracefully, with appropriate feedback to the user. This is achieved through the use of catch blocks in our asynchronous operations, along with Redux for managing the state of error conditions.

4.4. Back-end implementation

The back-end of our College Attendance Tracker is developed using Node.js [2] and Express.js [5], with the help of which we have managed to create an efficient and robust API that serves the requests that come from the front-end.

It handles various operations such as user authentication, attendance management, and user data manipulation.

4.4.1. Database and Models

For data persistence, we use MongoDB [3], known as a NoSQL database, since it has flexibility and scalability benefits.

We use Mongoose, an Object Data Modeling library for MongoDB and Node.js, because this library is useful in order to manage relationships between data, provide schema validation, and used to translate between objects in code and the representation of those objects in MongoDB.

When we check the database, it includes three main tables: attendances, courses, and users. Some of our primary models include: attendances, courses, and users.

- The attendances model looks like following:

```
const attendanceSchema = new mongoose.Schema({
  _id: {
    type: mongoose.Schema.Types.ObjectId,
  },
  courseId: {
    type: String,
    required: true,
  },
  groupId: {
    type: String,
    required: true,
  },
  date: {
    type: Date,
    required: true,
  },
  records: [
    {
      studentId: {
        type: String,
        required: true,
      },
      status: {
        type: String,
        enum: ["present", "absent", "excused"],
        required: true,
      },
    },
  ],
});

const Attendances = mongoose.model('Attendance', attendanceSchema);
module.exports = Attendances;
```

Figure 44. Attendance model

So, this model tracks the attendance records. The attendance has its own ID, the course ID, the group ID, the date that it has been created, and an array of records that has the student ID and their status (present, excused or absent).

- The courses model looks like this:

```
const courseSchema = new Schema({
  name: {
    type: String,
    required: true,
  },
  semester: {
    type: String,
    required: true,
  },
  professorId: {
    type: Schema.Types.ObjectId,
    ref: 'Professor',
    required: true,
  },
  course_date: {
    type: Date,
    required: true,
  },
  group: {
    groupId: {
      type: Schema.Types.ObjectId,
      ref: 'Group',
      required: true,
    },
    groupName: {
      type: String,
      required: true,
    },
    studentIds: [{
      type: Schema.Types.ObjectId,
      ref: 'Student',
      required: true,
    }],
  },
});

const Course = mongoose.model('Course', courseSchema);
```

Figure 45. Course model

Each course has the name, semester, the professor ID of the professor that teaches it, the course date when the course has been created, and an object of a group that has the group ID, the group name and an array of the student IDs.

- The user model looks like this:

```
const userSchema = new mongoose.Schema({
  _id: {
    type: mongoose.Schema.Types.ObjectId,
  },
  role: {
    type: String,
    required: true,
  },
  email: {
    type: String,
    required: true,
  },
  password: {
    type: String,
    required: true,
  },
  auth: {
    type: String,
    required: true,
  },
  firstName: {
    type: String,
    required: true,
  },
  lastName: {
    type: String,
    required: true,
  },
  birthday: {
    type: Date,
    required: true,
  },
  yearOfEnrollment: {
    type: Number,
    integer: true,
    required: false,
  },
  consultationHours: {
    type: String,
    required: false
  },
  course: {
    type: String,
    require: false
  },
  resetPasswordToken: {
    type: String,
    require: false
  },
  resetPasswordExpires: {
    type: String,
    require: false
  }
})
const Users = mongoose.model('User', userSchema);
```

Figure 46. User model

The user model is filled with the user ID, role, email, password, the user's first name and last name, their birthday, year of enrollment (used if the user is a student), consultation hours (if the user is a professor), course (used for professor too), the reset password token, and the reset password expire.

4.4.2. Authentication and Authorization

User authentication is handled using JSON Web Tokens (JWT). Upon successful login, the server generates a JWT with the user's ID and role as the payload and sends it to the client. The client stores this token and includes it in the header of subsequent API requests. The server verifies this token to authenticate the user for their requests.

Role-based authorization is also implemented. For instance, only professors are authorized to change attendance status, and only admins can manage user accounts.

We have functions that are called “studentAuth”, “adminAuth”, “professorAuth”, which can be found in the files that are called the same way as the functions. If we take a close look they do the same thing but differ by the condition. Depending on the role that is written on that condition, the user gets authenticated as.

In the following pictures we can notice what we said above.

```
function studentAuth(req, res, next) {  
  const authorization = req.headers.authorization;  
  if (authorization) {  
    const decode = jwtDecode(authorization)  
    if (decode?.role == "student") {  
      req._id = decode?.id;  
      req.email = decode?.email;  
      req.role = decode?.role;  
      next(); // call next() to move to the next middleware or route  
    } else {  
      res.status(401).json("You are not a student!");  
    }  
  } else {  
    res.status(401).json("You are not authenticated!");  
  }  
}
```

Figure 47. Function that authenticates the student

```

function professorAuth(req, res, next) {
  const authorization = req.headers.authorization;
  if (authorization) {
    const decode = jwtDecode(authorization)

    if (decode?.role == "professor") {
      req._id = decode?.id;
      req.email = decode?.email;
      req.role = decode?.role;
      next(); // call next() to move to the next middleware or route
    } else {
      res.status(401).json(["You are not a professor!"]);
    }
  } else {
    res.status(401).json("You are not authenticated!");
  }
}

```

Figure 48. Function that authenticates the professor

```

function adminAuth(req, res, next) {
  const authorization = req.headers.authorization;
  if (authorization) {
    const decode = jwtDecode(authorization)
    if (decode?.role == "admin") {
      req._id = decode?.id;
      req.email = decode?.email;
      req.role = decode?.role;
      next(); // call next() to move to the next middleware or route
    } else {
      res.status(401).json("You are not a admin!");
    }
  } else {
    res.status(401).json("You are not authenticated!");
  }
}

```

Figure 49. Function that authenticates the administrator

These files of authentication are found in the middleware folder of the back-end.

4.4.3. API Routes and Controllers

Our server-side routes and controllers correspond to the features of our front-end components.

Each route has corresponding controllers that handle the business logic for processing requests and interacting with the database.

```
app.use("/api/login", require("./routes/login"))
```

Figure 50. Accessing the Log In routes

```
app.use("/api/admin/attendances", adminAuth, require("./routes/admin/attendances"));  
app.use("/api/admin/courses", adminAuth, require("./routes/admin/courses"));  
app.use("/api/admin/users", adminAuth, require("./routes/admin/users"));  
app.use("/api/admin/profile", adminAuth, require("./routes/admin/profile"));
```

Figure 51. Accessing the Administrator routes

```
app.use("/api/professor/attendances", professorAuth, require("./routes/professors/attendances"));  
app.use("/api/professor/profile", professorAuth, require("./routes/professors/profile"));
```

Figure 52. Accessing the Professor routes

```
app.use("/api/student/attendances", studentAuth, require("./routes/students/attendances"));  
app.use("/api/student/courses", studentAuth, require("./routes/students/courses"));  
app.use("/api/student/profile", studentAuth, require("./routes/students/profile"));
```

Figure 53. Accessing the Student routes

4.4.4. Error Handling

Our back-end also includes robust error handling and request validation to protect against faulty or malicious requests. Errors are passed to Express.js's built-in error handler, which sends an appropriate HTTP response to the client.

We use the `express-validator` middleware to validate request bodies before they reach our controllers. This allows us to catch and respond to bad requests before they can cause any issues in our business logic.

4.5. Testing

To test our web application we use unit testing and manual testing. Running the unit tests was done using Jest [6], for both front-end and back-end.

Our unit test coverage was 70 percent, and the other 30 depend on user behavior, so they were tested manually. We applied the unit tests on both, the front-end and the back-end, in order to have a quality application.

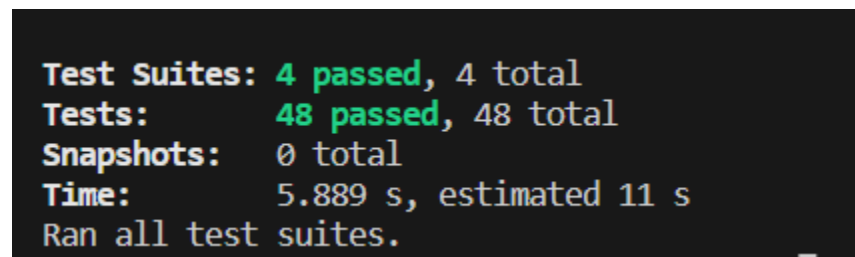
The test cases are stored into their corresponding test file, which can be found in the same folder where the component that is being tested is.

To test the front-end, we first have to get inside of the client folder using the `cd` command in the terminal. After getting in the folder, in the terminal we run:

```
npm run test-client
```

Doing this, we change the directory inside the client folder, and run the test cases that are tested using jest.

After the test cases are done being tested, we will get this in our terminal.



```
Test Suites: 4 passed, 4 total
Tests:      48 passed, 48 total
Snapshots:  0 total
Time:       5.889 s, estimated 11 s
Ran all test suites.
```

Figure 54. Passed test cases of the front-end

To run the test cases for the back-end, we navigate back to the main folder `cd ..`. When we are back there, in the terminal we run:

```
npm run test-server
```

This command, runs the test cases for the middleware folder and the routes folder using jest. After the test cases are done being tested, our terminal will show this:

```
> jest ./middleware && jest ./routes

PASS middleware/profesorAuth.test.js
PASS middleware/studentAuth.test.js
PASS middleware/adminAuth.test.js

Test Suites: 3 passed, 3 total
Tests: 9 passed, 9 total
Snapshots: 0 total
Time: 1.063 s
Ran all test suites matching /\.\\middleware/i.
```

Figure 55. Passed test cases of middleware

```
Test Suites: 10 passed, 10 total
Tests: 32 passed, 32 total
Snapshots: 0 total
Time: 13.821 s
Ran all test suites matching /\.\\routes/i.
```

Figure 56. Passed test cases of the routes folder

In conclusion, all of our test cases have been passed.

5. Chapter 5

Appendix

5.1. Glossary of terms

Git: A distributed version control system for tracking changes in source code during software development.

Repository (repo): A directory or storage space where your project lives. It can be local to your computer or a storage space on GitHub or another online host.

Clone: A Git command for copying a Git repository from a remote server.

Node.js: An open-source, cross-platform, JavaScript runtime environment that allows developers to run JavaScript on the server-side.

npm (Node Package Manager): The package manager for Node.js, which is used for installing, sharing, and managing dependencies in projects.

React: A JavaScript library for building user interfaces, primarily for single-page applications. It's used for handling the view layer in web and mobile apps.

Express.js: A web application framework for Node.js, designed for building web applications and APIs.

MongoDB: A source-available cross-platform document-oriented database program, classified as a NoSQL database program. It uses JSON-like documents with optional schemas.

Backend Server: The technology responsible for the server-side operations, database interactions, and server response to browser requests in a web application.

Frontend Server: Represents the presentation layer of the web application, interacting directly with the user.

5.1. Reference links and resources

[1] Git Download

<https://git-scm.com/download/win>

[2] Node.js and npm Download

<https://nodejs.org/en/download>

[3] MongoDB Download

<https://www.mongodb.com/try/download/community>

[4] React Documentation

<https://legacy.reactjs.org/docs/getting-started.html>

[5] Express.js Documentation

<https://expressjs.com/>

[6] Jest Testing

<https://jestjs.io/docs/tutorial-react>