

UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI
FACULTATEA DE INFORMATICĂ



LUCRARE DE LICENȚĂ

Laboratory Management Platform

propusă de

Daniel Chisă

Sesiunea: *Iulie, 2020*

Coordonator științific

Lect. Dr. Rodica Condurache

**UNIVERSITATEA “ALEXANDRU IOAN CUZA” DIN IAȘI
FACULTATEA DE INFORMATICĂ**

Laboratory Management Platform

Daniel Chisă

Sesiunea: *Iulie, 2020*

Coordonator științific

Lect. Dr. Rodica Condurache

Content

Introduction and Motivation.....	1
Contributions.....	3
I. Application architecture	
I.1 Overview.....	4
I.2 Storing data.....	6
I.3 Use cases.....	8
I.4 Application usage scenario.....	10
II. Application structure elements and functionalities	
II.1 Overview.....	12
II.2 Common functionalities	
II.2.1 Connection.....	13
II.2.2 Messenger service.....	16
II.3 Professors functionalities.....	18
II.3.1 Functionalities of common professors.....	18
II.3.1.1 Creating a new laboratory.....	19
II.3.1.2 Starting laboratory attendance and grading session....	20
II.3.1.3 Management of laboratories.....	23
II.3.1.4 Laboratories overview	24
II.3.2 Functionalities of lecture owners	25
II.3.2.1 Create a new lecture	25
II.3.2.2 Grading exams and assign extra points.....	26
II.3.2.3 Semester point of view of related laboratories.....	28
II.3.2.4 Lectures overview.....	30
II.4 Students functionalities.....	31
II.4.1 Profile configuration.....	31
II.4.2 Attend a laboratory session.....	32
II.4.3 Semester overview.....	34
III. Significant implementation details	
III.1 Overview.....	35
III.2 Significant details.....	35
III.2.1 Connection.....	37
III.2.2 Messenger service.....	39
III.2.3 Create a new laboratory or lecture.....	45
III.2.4 Laboratory sessions.....	48
III.2.5 Stored data overview.....	55
III.2.5.1 Professors overview.....	55

III.2.5.2 Students overview	59
III.2.6 General implementation details.....	60
Conclusions.....	64
Bibliography.....	66

Introduction and Motivation

Laboratory Management Platform is an online application hosted on Google Cloud servers at <http://labattend.appspot.com> designed to facilitate the management of the student activity from the faculty laboratories during the entire semester providing a set of functionalities for professors to create attendance and grading laboratory sessions for students to participate, to visualize, modify, centralize and export the situation of every student consisted of points, attendances and exams grades through a set of tools easy to manipulate. Also, the application encourages communication between teachers and students through the provided messenger service.

The application is divided into two sections, one for each category of users: the students and the professors, having a friendly and easy to use interface that provides a set of pages each corresponding to the main functionality linked together by the application menu that offers easy access to the application facilities including profile configuration, laboratory attendance, semester overview of recorded data, creating laboratories and lectures, starting laboratory sessions, exporting data and messenger service. More functionalities are described in the chapter of *Application structure elements and functionalities*.

The motivation of choosing this theme is represented by the need for students to have a single application that facilitates the recording of data in the same application and a way to have all the attendances, exams grades and partial points obtained during the semester synthesized and easy to view, also a way to conserve the environment by replacing the classic sheets of paper with the online attendance laboratory sessions. The development of a software solution that represents a better way for teachers to identify students through the profile picture and access to identification data as a student group and enrollment number. In addition, the ability for students to visualize their activity during the semester and not at the end of it, providing a way of awareness on the situation of each one that would represent an impulse for more involvement in the activity at the faculty.

The thesis has a topic approach which is topical as there are not many solutions in the computer science space, there are a few applications for attendance management-oriented only to a single lecture but not providing a solution that includes and facilitates the management of the student situations at every lecture students are registered.

This thesis aims to highlight both the structure and architecture of the application and the functionalities it offers along with the technical details of the development process based on the knowledge gained over the years of study and online information sources related to the new concepts used in the application.

The structure of the thesis consists of the chapters that contain the most important and synthesized information about the application grouped according to the concepts approached in order to offer a broad and overall image of both the current application and its development process.

The general objectives of the thesis are represented by the main chapters that have suggestive and significant names for their content and are linked together to provide a detailed point of view about the presented application including the architectural elements, functionality and implementation details from the development process.

The chapter *Application architecture* consists of a technical point of view and includes architectural details of the application, use cases, information about used technologies, the structure of the stored data and a usage scenario the users interact with.

The chapter *Application structure elements and functionalities* contains the description of the functionalities provided by the application considering the user hierarchy, the structural elements on which the application is built and contained in the user interface and the mechanisms that occur within the application when interacting with the user.

The chapter *Significant implementation details* rely on the main implementation details and solutions that solve certain problems within the application in order to confer the current functionality of the application. It also includes documented code examples from the implementation process.

Contributions

The application Laboratory Management Platform proposes a simple solution to use with maximum efficiency to facilitate the process of managing the activity of students for the entire semester and for any subject in a faculty being used by both teachers and students.

The purpose of the application is to translate the classic activity carried out on paper in the online environment to provide full accessibility to student information at any time during the semester. It provides the most important management functionalities for grades, points and attendance, the data is organized and easy to export.

The application facilitates the administrative work of teachers through tables containing information about students, data centralization, automatically generated and general calculations throughout the semester. Teachers can modify the stored information about students to remedy any mistakes in the grading activity very simply. The application proposes search engines for laboratories and students so that the access focused on one person is fast.

The basic activity of the application is represented by the attendance sessions that are started by the teachers, at which in real-time, the students can register and can be identified by personal information but also by the profile picture. Teachers can reward points for each lab to a particular student.

The application benefits students by facilitating the grading process in faculty by providing real-time results obtained once they are awarded by the teacher. Students can visualize from a broad perspective their current situation in a certain subject and can take measures to improve this situation if the student's grades are not favorable to the promotion of the subject.

The application facilitates communication between teachers and students through the messenger service offered by the application through which one can easily communicate about problems, errors. Service users are notified when new messages appear.

The application is developed based on modern Google Cloud technologies that offer a number of advantages as a simple pricing model and the cheapest on the market, offers robust data privacy and security features, performance, whether in terms of latency, speed, processing power or redundancy in networks.

I. Application architecture

I.1 Overview

The topic of this chapter is related to the architectural point of view over the application presenting the core design of the application along with the most significant details about the elements that form the application and used technologies. Also, it provides a clear image of user use cases, the actions that can be made, how to easily use the application described in the application usage scenarios, and a point of view about the data on how it is stored and accessed.

Application architecture is a map of how this software application is assembled as part of its overarching desired architecture and how elements interact with each other to meet requirements. Application architecture helps to ensure that scalable and reliable attributes are reached.

The application is created under the Google Cloud Platform, uses an App Engine service that ensures the build of a highly scalable application on a fully managed serverless platform. App Engine is a cloud platform for developing and hosting web applications in Google-managed data centers, according to the book from the *Bibliography* at index [7], Part III, App Engine: fully managed applications, and to the link found at index [3] *Google App Engine* from *Bibliography*.

Under Google Cloud Platform the application is a top-level container that includes the service, version, and instance resources that make the application. Resources are created in a certain chosen region, including the app code along with a collection of settings, credentials, and app's metadata. General information about the Google Cloud Network found at [4] *Google App Engine Overview* from *Bibliography* represented the source of information for this paragraph.

This application has an architectural pattern known as Model-View-Controller (MVC architecture) that separates the application into three main logical components: the model, the view, and the controller, information about this architecture are provided according to the source found at index [10] *MVC Architecture Overview* from *Bibliography*.

The Model component corresponds to all the data related logic that the user works with, representing the data that is being transferred between the View and Controller components and other logic-related data. All data is stored in Cloud by way of Datastore from Google Cloud Platform. Datastore is a NoSQL document database built for automatic scaling, high

performance, and ease of application development. Also, data as profile images are stored in buckets of Cloud Storage. Related information about Datastore and Cloud Storage can be found at index [1] in *Bibliography*, in the book at Part III, Google Cloud Platform-Storage Products, where it is presented how those services work.

The View component is used for all the user interface logic of an application and consists of all user interface components grouped as a set of HTML pages linked together to provide easy use to the user. Html pages contain Html code and are stylized with the help of CSS language. Each page is linked to a script that manages the flow of data the user sends to or receives from the server. Also, those scripts consist of Javascript, Ajax, and JQuery code, listen to the user interactions and manage dynamically parts of Html elements and render them. Communication is made by post requests to the server which sends responses needed by the user.

The Controller component acts as an interface between Model and View components to process incoming requests, manipulate data using the Model component and interact with the Views to send responses to requests. It consists of a Nodejs script that contains required code for the server and the endpoints routes that handle every request which comes from the View.

The architecture of the application based on Google Cloud technologies is composed of more services available in the cloud area that are well-defined, linked together to accomplish desired requirements. The main target was that application to be hosted and easy to get for its future users and with the help of App Engine, this can be done. A scratch of a common cloud architecture from website <https://cloud.google.com/solutions/web-serving-overview> is displayed in *Figure 1*:

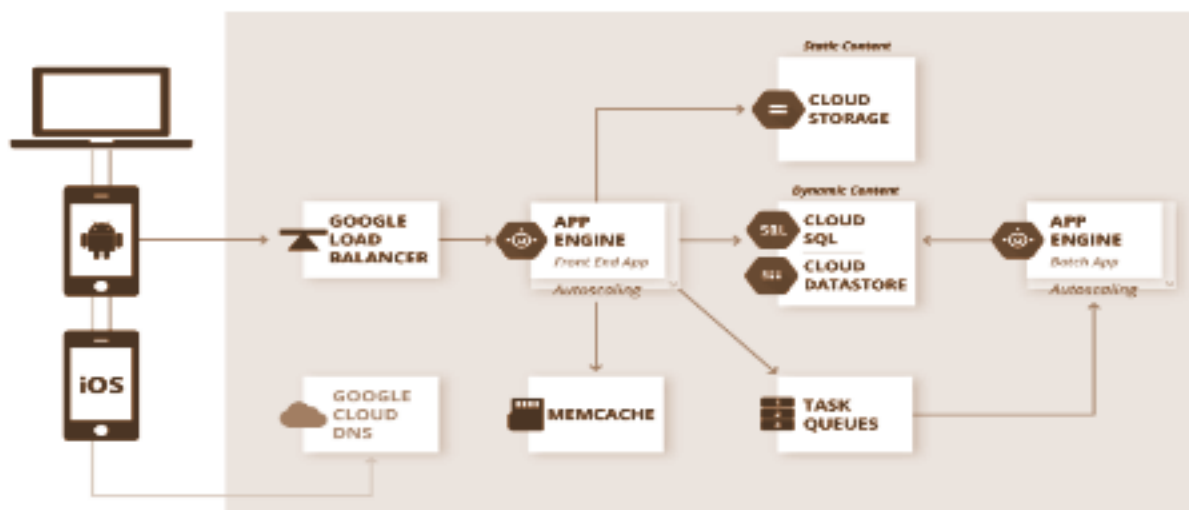


Figure 1: Architecture diagram of an App Engine Application

The development cycle includes several stages such as writing code, testing, deploying, managing, upgrading and building. Interaction with Cloud Platform in deploying stage is accomplished with Cloud SDK which includes the gcloud command-line tool.

The application lifecycle consists of the actions performed inside App Engine which respond to user's requests. A request starts from the user which contacts the application with an HTTP request. The application receives the request, identifies it from the domain name address, searches for a server that can provide the fastest response, sends the content of the request, receives the response and returns it to the user. The runtime environment begins when the request handler starts and is terminated when the request is done and the response is returned.

I.2 Storing data

An application like this one needs to store and receive its data very quickly because it needs a fast response from the server with the data needed to fill a user interface component as a table containing all students' status during an entire semester. According to the link found at [8] *Google Datastore Overview* from *Bibliography*, Google Cloud Platform has different services for storing data but the one we found efficient was the Cloud Datastore service designed to automatically scale to very large data sets, allowing our application to maintain high performance as they receive more traffic from the users. All queries are served by previously built indexes. The structure of the database can be visualized in *Figure 2*.

Users table store personal data of application users like userID which is an auto-generated key by Cloud Datastore needed to uniquely identify a user and store information related to the user necessary for the normal flow of the application. Firstname and Lastname are strings that help to configure the personal profile of users. Email and Password fields are the connection attributes that ensure the security and confidentiality of the application.

The student's table is designed for storing extra information for student users such as the group where a student belongs and registration code assigned by the faculty to each student. Those information helps with profile configuration and for better identification of students that attend laboratory sessions.

Lectures table is filled with the information given by the users that have the status of lecture owner, meaning that this user creates a lecture for which a code is auto-generated

(Lecture_Code) need for following laboratories creation under this lecture area by the same user or by other users that want to start attendance sessions. LaboratoryID belongs to the ID of a laboratory created by a user that wants to start a session for this lecture. OwnerID is the userID of the owner that created a lecture.

Laboratories table is related to Lectures and stores information about every created laboratory by professors and includes LaboratoryID needed to identify and store sensitive information related to a certain laboratory. ProfessorID is the userID of a user that has the status of a professor and represents the owner of a certain laboratory. The title field is a string that contains the name of the laboratory. Week_Number stores the current week from 1 to 14 needed in storing information about an attendance session on a certain week. Lecture_Code is the code of the lecture the laboratory is related to. Attendance_Code is the auto-generated code as a 5-digits code when users want to start an attendance session for a certain laboratory.

Records table is related to Laboratories and stores an entry for each student that enrolls in an attendance session. LaboratoryID field is the ID of the laboratory that a student enrolled in. StudentID is the userID of the user that is the student. Attendance field is a boolean type field which initially stores true value if a student participated in an attendance session of a certain laboratory, on a certain week. Points field stores the number of points that a student was graded in a certain session of a laboratory.

Exams table is related to Lectures and stores information about student's exams like the midterm exam, final exam and extra points on a certain lecture identified by Lecture_Code. StudentID is the userID of the student related to an entry in this table.

Messenger table stores data about the messenger functionality of this application and contains From_ID which represents the UserID of the user that sends a message to another user identified by To_ID which is the UserID of the user representing the destination person who receives the message. Seen field store boolean type data that represents if the destination person saw the message or not. The message is the text field that stores the body of the message a user sends to another user. Time is a date field needed to maintain chronological order of the messages at the moment that a user visualizes his conversation with another user.

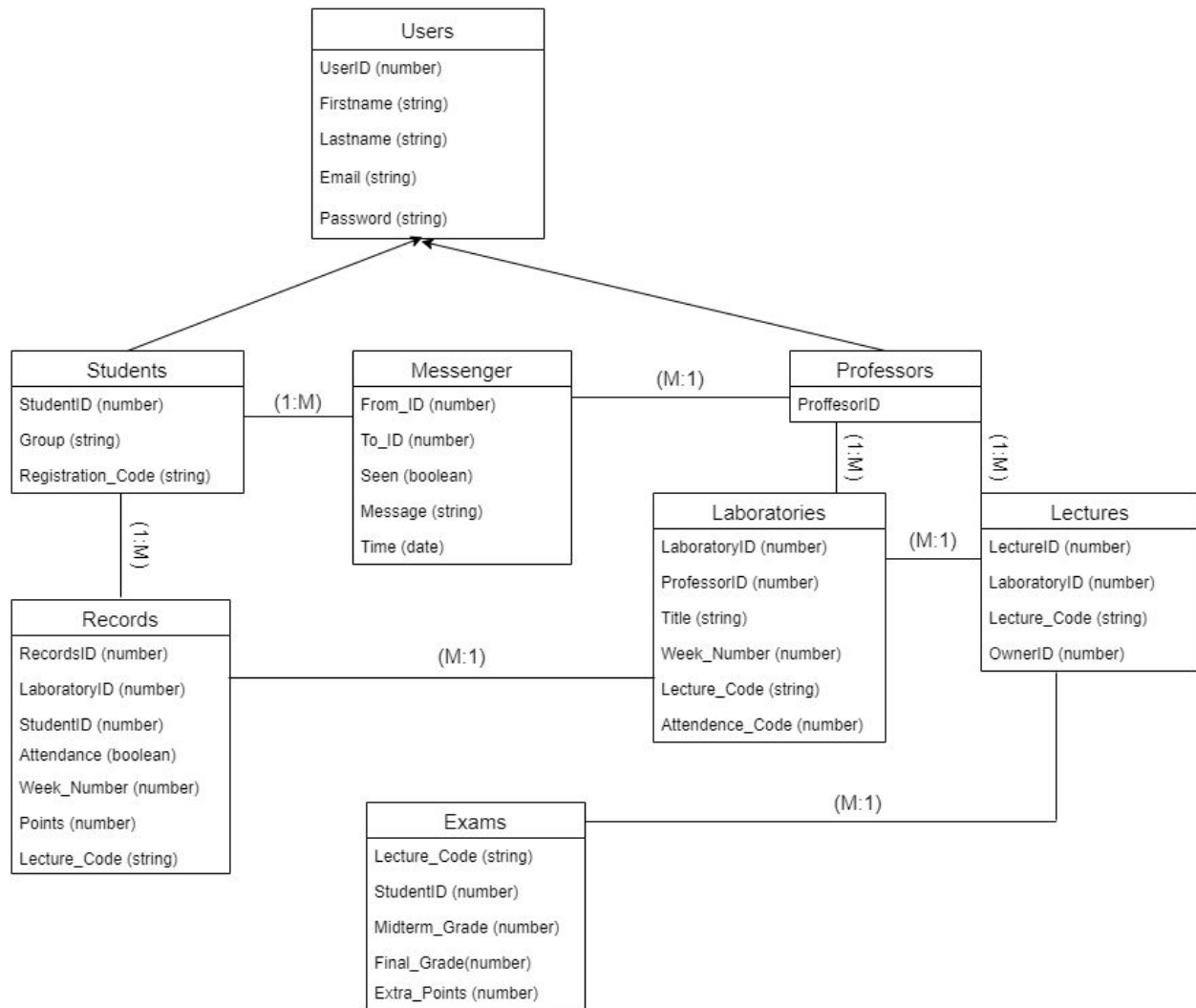


Figure 2: Database diagram

I.3 Use cases

The main actors of the application are the students and the professors which represent the set of persons who interact under the application area. The professors that use the application can have dual status as simple professors and lecture owners. Actors have a set of functionalities related to their role which can be observed in *Figure 3*.

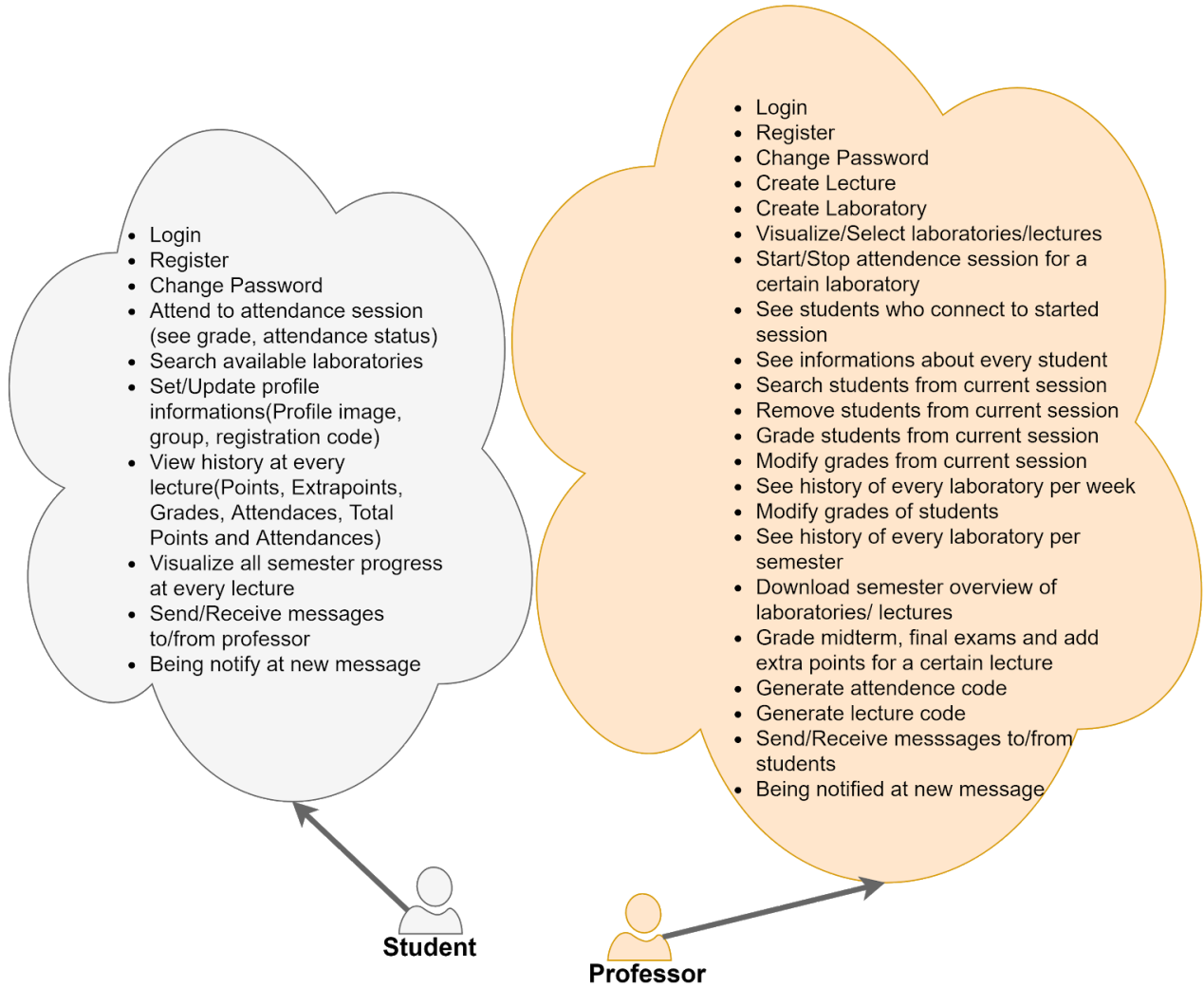


Figure 3: Use cases diagram

The simple professors are users which create a laboratory related to an already existing lecture and start attendance sessions. They have only access to the data about the students enrolled in their laboratories and the ability to manage session points and attendances.

The professors are the users that create and owe one or more lectures and have the management privileges of their lectures and related laboratories to them. They have abilities like grading midterm and final exam, give extra points to students, centralize all data about grades,

attendance and points along the whole semester, related to all students enrolled in a certain laboratory under their lectures domain.

The simple professors and owners of the lectures are those users that can start an attendance laboratory session on a certain lecture domain on a certain week.

The students are the users being the persons enrolled in a certain attendance laboratory session on a certain week during the real laboratory that takes place at the faculty under the management of a professor.

I.4 Application usage scenario

The application represents a tool that has the purpose to facilitate a better organization and easy access to elements that define the status of student's activity along the semester during the laboratories they participate in. It provides mechanisms to support an real-time online interaction between professors and students entitled attendance and grading laboratory sessions. Also, the students are able to see their activity history in an easy to use way provided by the application. All the users have to be registered, to have a valid account to have access to the application services.

Attendance and grading session stands for an online activity during the time of the laboratory who takes place in real life at faculty related to a certain lecture. The professors are authorized to start sessions by selecting the name of an already created laboratory, generate an attendance code and provide it to the target students who will attend the session, choose the week in the semester at the time the laboratory took place and then start the session. After the session is started the students can attend by selecting the name of the laboratory and entering the attendance code. If the attendance is successful meaning that the attendance code is valid, the students have automatically recorded the status of attending a certain laboratory. The professors can grade the student activity during the laboratory and the students are notified about the grades.

The recorded activity of the students represents the main data managed by the application and can be accessed using application services to visualize the history of every laboratory session and to modify graded points by the professors who graded them. The professors are able to download the stored data about the student activity, to grade the exams of the students and to give extra points to the students if the case. Students are able to monitor their activity during the

semester in the laboratories they participated in. Both students and professors can communicate using the messenger service from the application.

More information about the user's capabilities can be found in the use case diagram provided in the *Use cases overview* section and the functionalities that extend those capabilities are detailed in the chapter of *Application structure and functionalities*.

II. Application structure elements and functionalities

II.1 Overview

The chapter focuses on information regarding the structural elements of the application that compose the user interface, the structural organization of the files and particular elements that serve the functionalities provided to the users in order to confer the designed experience and the usage behavior on the platform. The application is divided into two sections: the student's section and the professor's section. Professor can be of two types: common and lecture owners. Functionalities are structured on three categories: common functionalities containing the same behavior for both users category, the student side and the professor side on each side the functionalities being different.

Every set of functionalities presented contains main details such as user capabilities, structure elements that compose each page and application mechanism flow that occurs behind the user interface described to provide a better view and understanding of how the application works.

This chapter contains important details regarding the structural elements that compose the application and the main functionalities derived from every user interface page that application provides to the users in order to achieve the designed behavior and to serve a good experience.

The structure of the application is based on Html pages that have a CSS code for a better styling content provided to users and backend code of javascript and ajax to manage the content from the user interface and send and get data from the application server in form of requests.

The server code is written in node js and includes all the logic of the application as endpoints treatment, session manager, server configuration and useful atomic functions with a high level of cohesion being designed to solve a certain problem. In most cases, those functions communicate with the database to perform operations needed by the application and serve the information to the users. Those functions will be mentioned in the context of presenting all served functionalities related to the application. Related to server script there exists a model script that contains the main functions that manage the communication between application and

database. This communication is made at every user session, the application has included credentials for accessing the database.

According to the information found at index [6] *Google Cloud Authentication* from *Bibliography*, every request made by the application must be authenticated so the users can access stored data under the Google Cloud Platform. Google Cloud APIs only accept requests from registered applications, which are uniquely identifiable applications that present a credential at the time of the request. Requests from anonymous applications are rejected.

Accessing private data on behalf of a service account outside Google Cloud environments is made with a service account key which is a private key as a JSON file. This file is passed to Cloud Client Libraries, so they can generate the service account credentials at runtime for Datastore and Storage services. Cloud Client Libraries will automatically find and use the service account credentials by using the `GOOGLE_APPLICATION_CREDENTIALS` environment variable. Private data consists of every information stored inside the database as grades, attendance, personal information.

Accessing public data such as profile pictures is made with an API key that only identifies the application and doesn't require user authentication. It is sufficient for accessing public data.

Application is divided into two parts and has a dual-mode of usability related to the duality of the users: the student mode and the professor mode. Each mode corresponds to a different area that has its own functionalities and structure depending on the role of the users and their activities which are different for both students and teachers.

The content shown on every usability mode is structured on Html pages representing the users interface and differs from student area to professor area. It is based on Html elements and scripting code to give to the users the planned behavior.

II.2 Common functionalities

II.2.1 Connection

The login page purpose is for users to choose the mode they want to use the application by selecting student or professor, then begins the login form to ensure the security of the

application by having authenticated users. After a successful connection, the resources are loaded for students or for the professor depending on which selection was made by the user. It is a common interface for both user types and provides the abilities of login, register a new account and to change the password if forgotten.

The page can be visualized in *Figure 4* and is composed of a switch between student and professor which brings to the user the corresponding login form, registration and forgot the password for each user's category. The *Login* button has an event listener set to catch the event when the user clicks to login. Before sending the input to the server it is checked to be valid data and required fields completed.

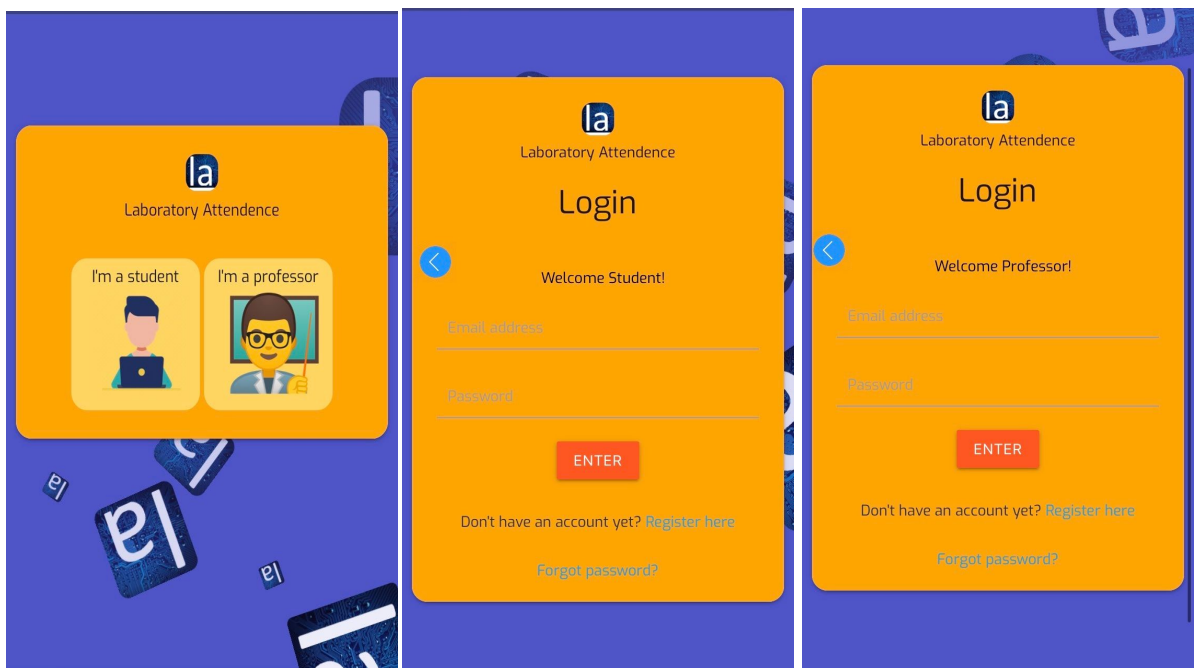


Figure 4: Set of login screenshots of the application

Another element is the blue back arrow that represents a button that helps the user to go back to the switcher page. The welcome message is adapted to the choice made by the user.

Login functionality is a matter of security that ensures the protection of data and actions made by the registered users of the application to be consistent, well-formed and with good intentions because the application provides an environment that manipulates sensitive data and actions that has an increased credibility degree. The authentication of the users is made based on the connection data stored by the application for each created account as email and password. This is the tuple that ensures the credentials for accessing the application environment. The user

has to fill the login form, the data entered are verified to be not empty and the email address and the password to exist in the Users table from the database. According to this check, the user is redirected to the main application or receives an informative message about the possibility of entered data to be wrong. With the help of javascript and ajax code placed in the backend, the input is sent to the server as a post request where the database is accessed and the login data is searched. The server responds with certain status codes according to the cases met about valid or wrong login credentials.

There are some images containing the logo of the application that are animated, moving around in the background to create a dynamic effect made with the help of animation attribute of CSS language by styling an Html list of a few elements that contains an image with the logo and continuously changing their sizes

The registration page is a form that has fields to be completed by a user that wants to create a new account being used in the future to access the application. Those fields are first name, last name, email, password. Before sending this data it is verified if the data to be registered into the database are valid and all the required fields are completed. Also is made a check for the email address to not be used before. After successful registration, the user is redirected to the login form.

Forgot Password page is a form that requests the user for the email address linked with the account for which the change password action is requested. Once the email address is a valid one, the application sends an authentication code to that email address. The user that wants to change his account password has to check his email inbox to get the code. That code has to be entered along with the new password in order to change the password to be done.

Logout functionality accessed by the click action on the button that has this meaning placed on the right top side of the user interface produces the destruction of the current session created at the moment the user successfully logged in.

II.2.2 Messenger service

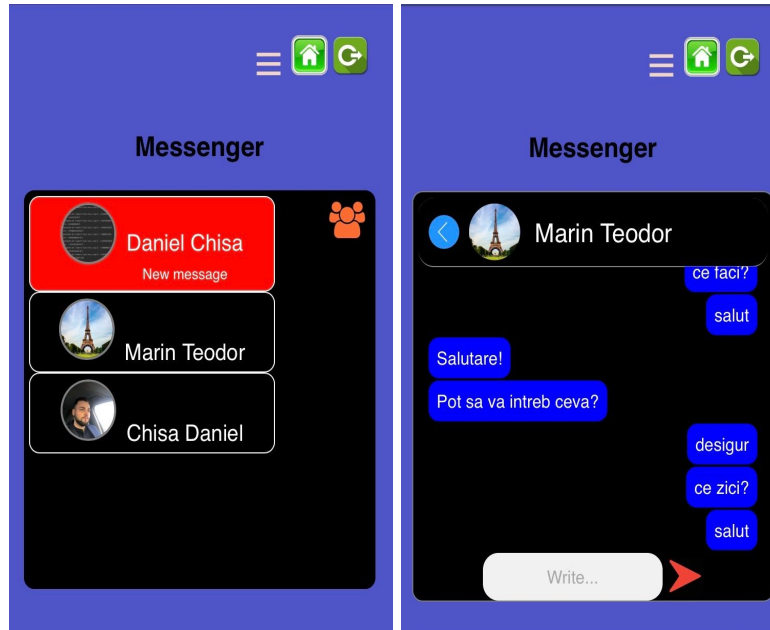


Figure 5: Set of screenshots of messenger service

The page containing the messenger service provides to the application users an easy communication between professors and students who interact in a laboratory session meaning if a professor starts an attendance and grading session and a student attends to it, therefore they can send messages to each other. This functionality was implemented to facilitate problems of misunderstanding or wrong grading issues that do not reflect the user's intentions. This is a way to notify a professor if the grade of a student gained during an attendance session was different in reality versus the platform stored data. The interface of the messenger service can be visualized in *Figure 5*.

Inside the database exists a table that records every message sent by the application users. It is also stored the ID of the person that sent the message and the ID of the person that will receive a message, also if the message has been seen by the last person and the time of sending. The mechanism behind the messenger will be described as follows. When a user enters the application or enters the page reserved for the messenger service the application sends a request to the server to search in the Messenger table from the database for entries where the id of the current user is stored in the position of the destination person in the table. The server processes

the result of the query and selects uniquely ids of users that sent messages to the current user. For each UserID are performed searches in the Users table to get the name of those users being sent in the Messenger page and being rendered. This response has a JSON form and along with the names, there is received the status of the seen column from the messenger table which helps to classify the users that send messages that were not seen yet and the seen messages. This distinction helps to notify users of new incoming messages along with all the pages where they navigate at a certain moment. The new messages received in the messenger will be red highlighted containing a distinctive notification.

The list of persons is dynamically constructed and every item of it has an event listener that listens for the click on a certain item action. When an item is clicked the server is notified to send as a response to the request containing the ID of the selected user and the current ID of the user, the conversation between them. The server performs queries in order to get from the database the messages from those two users identified by their ids and construct a JSON response which includes the messages between the users and who sends each message in order to render correctly every message in the user interface to the left if the message was received or to the right if the message was sent by the current user. The data from the response is sorted regarding the time of sending in order to accomplish a chronological order of the messages. Once the response is received by the application the conversation is shown along with the name of the user that sends messages, his profile picture, the back arrow to return to the list of persons, the input and the *Send* button who facilitate the action of sending messages.

The function executed by pressing the red arrow turned to the right button checks if the message text box is not empty and sends the message along with the sender and receiver IDs to the server that adds a new entry in the messenger table. Once the adding is done the conversation of the sender of the message is updated.

The notification system that notifies a user when a new message appears consists of a cyclic request to the server to send the updated conversation to the user from time to time to simulate the real-time speaking. When the cyclic response from the server is received and contains the updated data, both conversations and the people list are updated on the page. If there exists some new messages sent to a user this user will be notified at the top right corner of the application layout, over the messenger icon that will have a red notification circle wrapping a number that represents the counting of the new unread messages.

The messenger also provides a search engine to its users to easily find the related users to speak with only if they are linked in the same session of the same laboratory. This can be done by pressing the *Red People* button from the first window that also contains the people list.

II.3 Professors functionalities

This area contains all the functionalities for the professors grouped in order to provide easy access and usability of concepts and information. The user interface is formed of page layout which wraps the characteristic elements of a certain page, the navigation menu which links all the pages, the *Messenger* button which redirects the user to the messenger page and also notifies the user when an incoming message appears, the *Home* button that redirects the user to the main page which is the starting session page and the *Logout* button which redirects the user to the login page.

The menu is a popup menu that is shown when the user clicks the *Menu* button and disappears when the *Menu* button is clicked again. It is structured relative to the duality of the professor's status to be a simple professor or a professor who also is a lecture owner. The menu is split into two sections: teacher section and lecture owner section.

The teacher section provides functionalities for common use such as creating a new laboratory, starting an attendance session, manager of laboratories and laboratory overview which includes a semester point of view about a certain laboratory.

The lecture owner section provides functionality for the user that owes a lecture under which will be created new laboratories to provide the ability to start attendance sessions. This menu includes capabilities of creating a new lecture, grading students registered on a certain lecture and an overview of registered laboratories under a certain lecture providing the name of the users who has the status of the owner over those laboratories and the lecture codes for each lecture that helps other users to create new laboratories under the domain of a lecture. Also, a semester point of view about all laboratories and the final situation of every student.

II.3.1 Functionalities of common professors

This area of the application refers to the functionalities provided to the common professors that own laboratories related to lectures owned by other users known as lecture owners. Those

functionalities give abilities to the user to manage attendance and grading sessions with the help of a friendly interface and easy access to the data and manipulate the application functions. A common professor can create lectures and become both a common professor and lecture owner. There is no restriction regarding users to belong just in one category of professors. Important fact is that a common professor cannot create a new laboratory related to a certain lecture without the lecture code provided by the lecture owner. The capabilities of the common professors will be presented below.

II.3.1.1 Create a new laboratory

This page can be visualized in *Figure 6* and provides to the professors the functionality of creating a new laboratory to have the ability to start attendance and grading sessions. The laboratory will be created under the domain of a certain lecture already created by a lecture owner user.

The page consists of a form that has 3 fields to be filled by the user corresponding to the name of the new laboratory that wants to be created, the group of students for whom the sessions will be created in the future and the lecture code is given to the users that want to create a laboratory by the lecture owner. This code is a primary key that relates to the new laboratory under the domain of a certain lecture to provide to its owner to manage an overview of the records that will be made for this laboratory. The code input has attached an on-key-up event function that sends to the server the code and gets the name of the lecture link by this code and the lecture name field is filled with this information.

To facilitate the usability of creating a new laboratory. Once the *Create* button is pressed the fields of the form are checked to be valid then the server receives the data to store for the newly created laboratory.

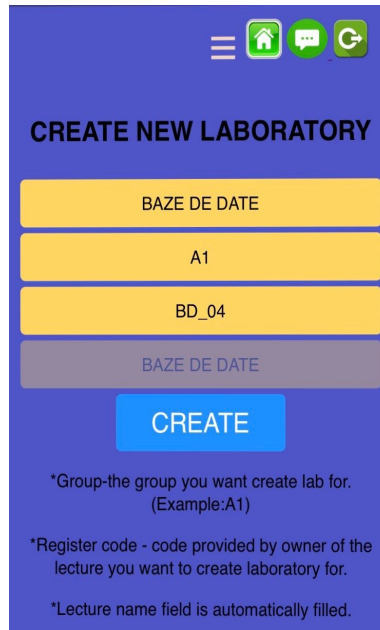


Figure 6: Screenshot of create new laboratory page

Before storing the data, the server query the database to check if there already exists a laboratory for this student group. After the creation, every new laboratory has an auto-generated laboratory code consisting of the acronym of the laboratory and the group of the students. Once a laboratory is created it is linked in the database under the lecture it holds by inserting a new row in the lectures table along with the generated laboratory code. After the creation is completed, the user is notified about the status of the operation, in the case of success, he is allowed to start attendance and grading sessions for the newly created laboratory.

II.3.1.2 Starting laboratory attendance and grading session

On this page that can be visualized in *Figure 7*, the application provides functionality to its users to start a new laboratory attendance and grading session for an already created laboratory. The user has to type the name of the laboratory or to easily select one from the

existing laboratories created by him, to generate code by pressing the *Get Code* button who auto-generates a 5-digits code, then to type the week for the session.

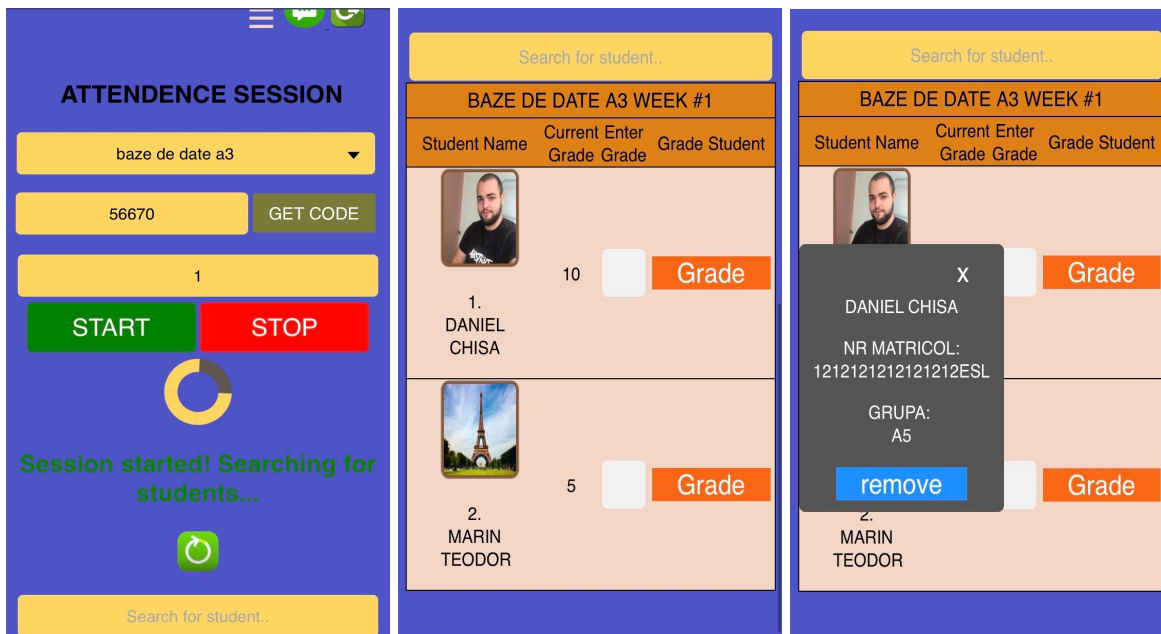


Figure 7: Screenshots representing attendance session

Pressing the *Start* button makes the session to be started and the students are allowed to attend by providing in their platform the generated code given by their teacher. The application automatically starts to update itself to facilitate and reduce user interactions, but there exists a refreshing button on the student's list that attends to the current session who provides faster updates over the data. The cyclic update is based on a cyclic request to the server to send a response with the current students that attended the session.

The list of students is dynamically auto rebuilt on javascript and consists of an Html table that includes the name of the laboratory, the week of the current session, students names, current grade if given, else a blank space, the grading box which is an input to type the points obtained by the student during to the session, the *Grade* button to grade the student. When the *Grade* button is pressed, it is sent a request to the server containing the grade and the ID of the graded student in order to record into the database the given grade. The application automatically updates those changes. Also, the students contained in the student's list are alphabetically sorted and an order number is given to easily have the count of the registered students.

In order to provide better recognition of the students that are registered to a current session, the popup box that includes more information about a student such as registration ID

given by the faculty to each student and the group of that student by pressing his cell in the table. It includes a button to give the ability to the professors to remove a certain student who connected to this session by mistake. Also, the profile picture shown for each student represents a good way for the teacher to better recognize the students that are in the class. Those profile pictures are uploaded dynamically from Cloud Storage who provides the Bucket service for storing files and easy access to data.

This page provides the functionality of a search bar to easily access a student row in the table. This search consists of a parse over the rows of the table and comparing the search text with each text contained in the field name. For the rows where the condition is true, the rows are shown and the rows where the condition is false are set to be hidden. This search bar is an Html element called input that has assigned an event listener function on key up that computes the described algorithm from above.

Pressing the *Stop* button generates the application to stop cyclic updates of the new students. Therefore, on this stage, the students are not allowed to attend anymore. However, the professor can grade the current students from the list, can visualize more information about everyone, and also has the ability to remove a student from the list.

When the page is loaded the necessary resources are loaded from the database such as the names of the laboratories owned by the user to provide the functionality to search a laboratory or to select one from the drop-down list. Once the *Start* button is pressed, the input fields are verified to contain valid data, a request with this data is made to the server who stores the attendance code and the week number in order to allow the students to attend. While the session is running meaning that it is not stopped by the user interaction of pressing the *Stop* button, it receives updated data from the server to provide real-time updates of the student's list. The cyclic updates are made by a function that has set an interval to call the function periodically.

Also, the cookies are used to store the input data and to preserve the current status of the session in the case the user navigates on other pages of the application. Therefore, if he leaves the current session, when he comes back the session will be restored as a frontend point of view. When he surfs on other pages of the application, the session remains open and the students are allowed to attend and the cyclic updates are made.

II.3.1.3 Management of laboratories

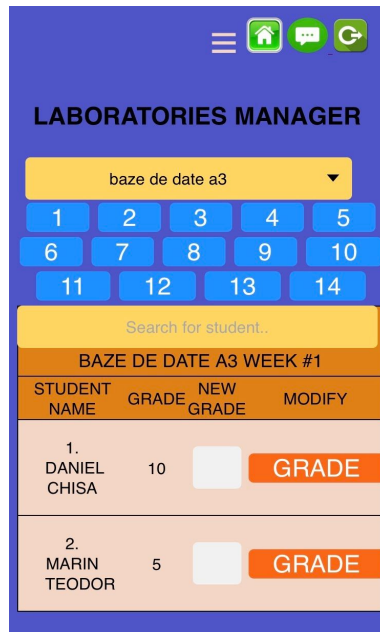


Figure 8: Screenshot representing laboratories manager

This page can be visualized in *Figure 8* and provides functionality to the common professors to easily manage their laboratories after the sessions are ended to overview the laboratory status grouped on each activity week. They have the capability to see information about each student with the help of the popup box which contains more information about the student being accessed by pressing the cell of a certain student, to modify the given grades during the passed laboratory session, to search for a student in the list of attendances or to remove a student from the list.

The structure is defined by a search bar where the users can find one of their laboratories they want more details for or select one from the drop-down list of laboratories, a set of buttons that represents the week number they want informations for, a search bar which filters the students and the table that contains the students from a certain laboratory session which took place in a certain week.

The list of students is alphabetically sorted, the students are counted and the modification of a grade regards in sending the new grade and the ID of the students to update the row in the record table from the database. Once the update action is done, the server sends the response and the grade is updated in the user interface.

II.3.1.4 Laboratories overview

**LABORATORIES
SEMESTER VIEW**

baze de date a3 ▼

SHOW

Search for student..

BAZE DE DATE A3																			
STUDENT			WEEK NO														TOTAL		
NAME	NR_MAT	GROUP	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14	TA	TP	
1. DANIEL CHISA	12121212121212esl	A5	1	10	0	0	0	0	0	0	0	0	0	0	0	0	0	1	10
2. MARIN TEODOR	100000000000ESL	A6	1	5	0	0	0	0	0	0	0	0	0	0	0	0	0	1	5

↓ Excel Download

Figure 9: Screenshot representing laboratories overview page

This page can be visualized in *Figure 9* and provides the functionality to visualize the laboratory status of every student registered along the semester on the attendance and grading sessions of a certain laboratory owned by the user that makes the search.

Professor can select or search one laboratory he wants more information for and by pressing the *Show* button a request is sent to the server-side and respond with the necessary data for dynamically building the main table that contains more details about every student. Before the request is made there are verifications made to validate the input field. The server performs queries in order to deliver data constant of the students registered on a session matching received parameters such as laboratoryID and professorID.

Information that belongs to the table consists of student name, registration code assigned by faculty to each student, the student's group to which the student belongs, the status of attendances and points during the semester of every weekly session. The response that contains those details is received from the server as JSON then is stored and parsed to extract the information in order to render them on the table. The total area computes the sum of the attendances and given points to facilitate a better view of the situation of the students. The laboratory owner has the ability to search a certain student from the table to better identify his situation and easily access the data, also to export the data by download as excel file the content of the generated table.

II.3.2 Functionalities of lecture owners

The area of lecture owners consists of providing functionalities to the users that create a lecture and becomes the owner of a certain lecture for which in the future other users known as common professors can relate new laboratories under a lecture domain. A lecture owner has access to the status of the related laboratories to his lectures, can visualize the situation of students registered on attendance and grading sessions along the whole semester, has access to the lecture codes and the owner's name of every laboratory. A new ability is to grade the midterm and final exam and also to add to each student some extra points. The lecture owner has the responsibility to provide the lecture code to the professors that want to create new laboratories related to his lectures. The capabilities of the lecture owner will be presented below.

II.3.2.1 Create a new lecture

This page can be visualized in *Figure 10* and provides functionality to the lecture owner to create a new lecture in order to give the capability to common professors to create new laboratories related to created lectures. The structure of the page consists of a form that has fields for the name of the lecture, the acronym of the lecture and a field that contains the auto-generated lecture code being composed of the acronym and a random 2-digits number. This code is required whenever a user wants to create a new laboratory related to a certain lecture





linked by the code. The functionality of creating a new lecture is made by a request to the server who query the database to verify if there is no other lecture with the same lecture code then insert a new row in the lectures table then send a response with the status of the action performed and the user is being notified about the newly created lecture. Before sending the request to the server, the data from the fields is verified in order to be valid.

Once a lecture is created, anyone who owns the lecture code can create laboratories and start attendance and grading sessions.

Figure 10: Screenshot representing create a new lecture page

II.3.2.2 Grading exams and assign extra points

This page can be visualized in *Figure 11* and provides functionality to the lecture owner to grade the students registered to sessions that belong to laboratories related to a certain lecture. The user can either search the name of the lecture or select one from the drop-down list from the search bar. By pressing the *Show* button the server receives the request and sends a response with the required data needed to fill the form. Received data is parsed and used to render useful information about students and currently obtained grades. With the help of javascript, the Html div who wraps the table is filled with a dynamic constructed table that contains the information received from the server.

STUDENTS GRADING

BAZE DE DATE ▼

SHOW

Search for student: .

BAZE DE DATE												
STUDENT			Grading						TOTAL			
NAME	NR_MAT	GROUP	Extra points	Set	Midterm Grade	Set	Final exam grade	Set	TA	TP	Midterm	Final Exam
1. DANIEL CHISA	12121212121212esl	A5	7	set	8	set	9	set	2	24	8	9
2. MARIN TEODOR	100000000000ESL	A6	0	set	3	set	2	set	1	5	3	2


 Excel Download

Figure 11: Screenshot representing grading students page

The table contains names of the students that attended the lecture laboratories ordered alphabetically and the registration code assigned by the faculty to each student and the student's group where the student belongs. Among this information, there are the current extra point, midterm and final exam grades, also a total of attendance and points.

The lecture owner has the ability to search a student from the table to easily identify his row and grade him. Grading action is made by entering a value in the associated input field and pressing the *Set* button next to it. When the *Set* button is pressed the application checks if there is a value in the associated input field to be updated in the database. The request contains the ID of the student that has to be graded, the value of the grade and the option that has to be updated: extra point, midterm, or final exam. Once the action of updating a field is done in the database, after the successful response of the server, the value is also updated in the user interface to ensure that the value was updated and stored. The total points column represents the sum between the points collected by the student during grading sessions of related laboratories and

the extra points given by the lecture owner. He has the ability to download the data inside the table and modify grades or extra points on how many times he likes, there is no limit who constrains this functionality.

II.3.2.3 Semester point of view of related laboratories

This page provides the functionality of visualizing the status of the related laboratories to a certain lecture. The interface can be visualized in *Figure 12* and includes details about the situation of each student during the semester attendance and grading sessions. It consists of two overviews based on a laboratory point of view and the lecture point of view which are generated after pressing one of the two presented buttons.

The overview based on laboratories related to a certain lecture provides a complete situation of each session that took place in the past under the lecture domain during those fourteen weeks of activity. Details regarding the student's name, registration code assigned by faculty to each student, the student's group where the student belongs and the tuples of obtained points and attendance record in each week. Total columns contain the sum of all the points and the sum of all attendance points. This kind of overview is accessed by pressing the *Labs* button.

The overview based on the lecture point of view centralizes all students' records that attended to attendance and grading sessions during the whole semester either to only one laboratory or to more than one laboratory. The application renders data such as a student who attended more than one laboratories related to the same lecture to have the situation centralized and complete by putting together all the existing data recorded during the semester. This kind of overview can be accessed by pressing the *Final* button.

Data contained in the table can be visualized in *Figure 13* and is about general student information such as name, group, registration code along with semester situation and the total columns that include a sum of all the attendances, total points consisting of the sum between bonus points and collect points from the sessions, also the grades for the midterm and final exams. Therefore, this overview contains all information needed to describe a complete situation about each student.

Those overviews are constructed dynamically in javascript with the received data from the server after ending of the initial request to the server. Both overviews support the export of the data contained in the tables and can be downloaded as an excel file by the lecture owner

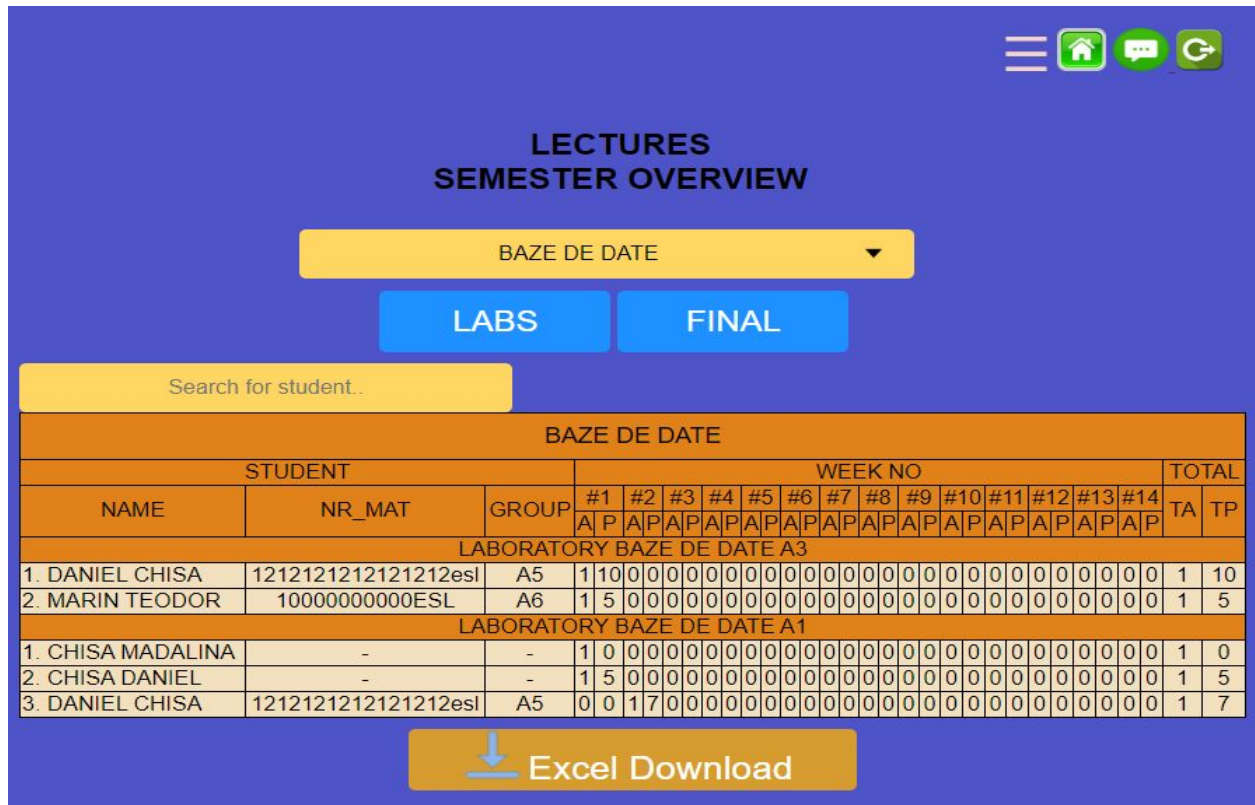


Figure 12: Screenshot representing semester point of view related to laboratories

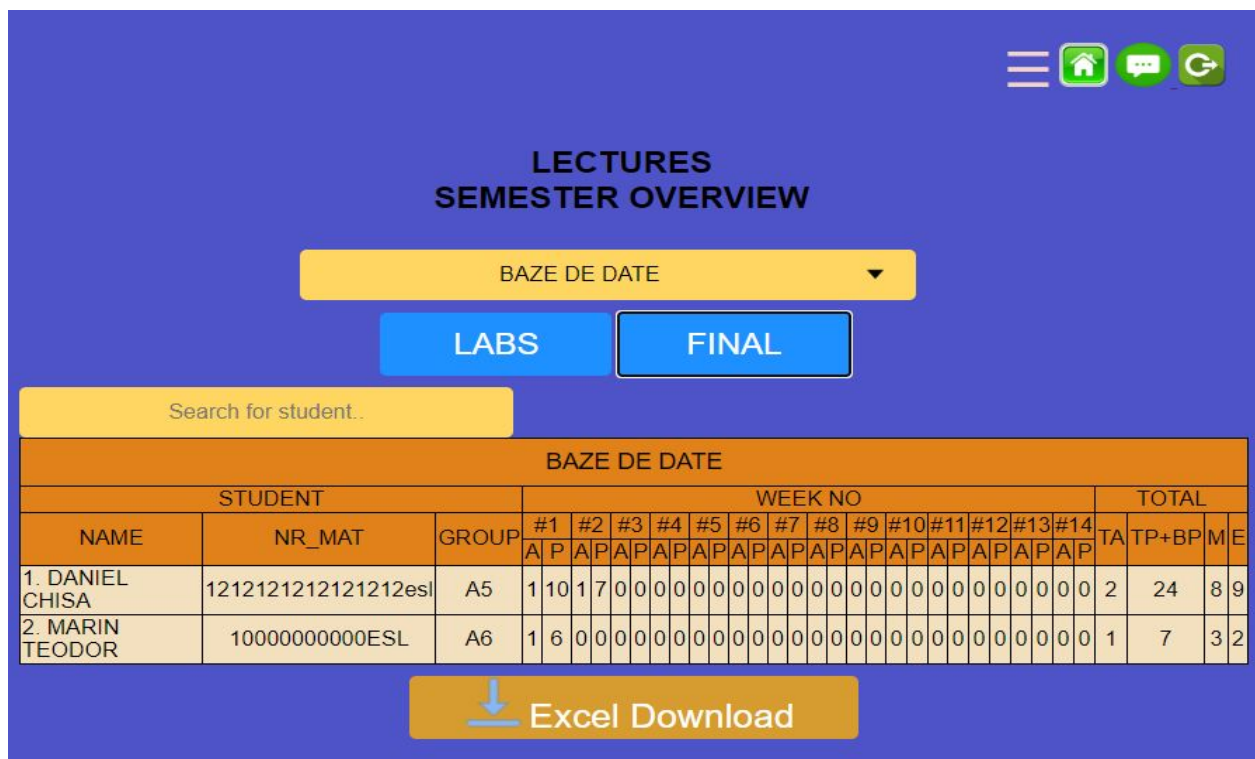
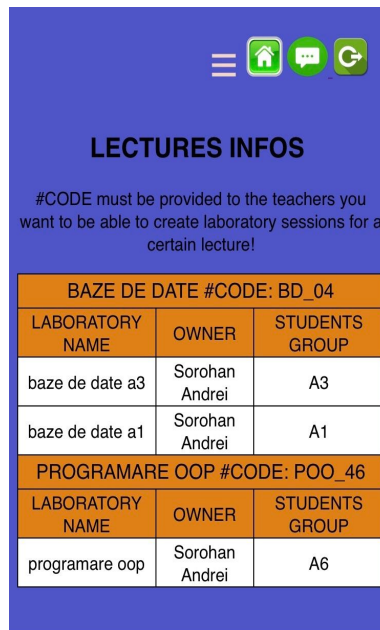


Figure 13: Screenshot representing based on the lecture point of view

easily with the integrated download button. Search over the tables is supported, therefore, the user can easily get a certain row from the table and fast check the situation of a student.

II.3.2.4 Lectures overview



BAZE DE DATE #CODE: BD_04		
LABORATORY NAME	OWNER	STUDENTS GROUP
baze de date a3	Sorohan Andrei	A3
baze de date a1	Sorohan Andrei	A1

PROGRAMARE OOP #CODE: POO_46		
LABORATORY NAME	OWNER	STUDENTS GROUP
programare oop	Sorohan Andrei	A6

Figure 14: Screenshot representing lectures overview page

This page provides an overview of all lectures owned by a lecture owner which consists of general details about each lecture, related laboratories to lectures, the name of the users that own laboratories and the students group for what a laboratory took place. Those pieces of information are rendered in a dynamically constructed table that is created when the user starts this page. A request is sent to the server who provides data about lectures owned by a user identified by his unique id. The utility of this page is to put together with general information easily reached by the owner to have a perspective of his lectures. Every lecture has an auto-generated code from the creation of a new lecture which is shown on the page to avoid users to forget the code needed for creating new laboratories related to lectures. The interface can be visualized in *Figure 14*.

II.4 Students functionalities

Students area is the side of application which refers to the users that are students registered on a faculty and provides functionalities such as profile visualization and configuration, attendance to laboratory sessions, communication with professors on the messenger and semester overview of the personal situation with plenty details about obtained points, attendances and exams grades.

The structure of the student's area is the same as the professor area, all the Html pages having the main zone where the specific information is wrapped and the navigation elements such as navigation menu, *Home* button, *Logout* button and *Messenger* button. The *Home* button is linked to the laboratory attendance page and the *Messenger* button to the messenger page.

II.4.1 Profile configuration

This page provides a personal view of identity information about a student containing the student name, registration code assigned by faculty for each student, the group where the student belongs and a drop-down form actioned by the *Set Infos* button for updating information such as registration code or the group name of the student. The interface of this page can be visualized in *Figure 15*.

The profile picture can be easily uploaded on this section either from the computer or phone and is automatically stored in the Cloud Storage Bucket from Google Cloud Platform and can be updated anytime. When the *Plus* button is pressed, the dialog for search and select an image is shown, after pressing upload the image is sent to the server then is stored as a file having the name of the student id. This bucket is open-source storage and the items stored can be accessed without authentication by the image URL. Therefore, the places where the profile picture is used consists of setting the URL attribute of an Html tag called image to the link of the corresponding image of a user identified by id.



Figure 15: Screenshot representing student profile page

II.4.2 Attend a laboratory session

This page provides students the functionality of easily attending a laboratory session by the name of the laboratory and the laboratory code given by the professor. Once the *Attend* button is pressed the input fields are checked in order to contain valid data, if the attendance code for the selected laboratory is wrong the student will be notified.

When the *Attend* button is pressed and the input data is valid is sent a request to the server with the data then the server checks the attendance code received and compares it to the attendance code stored in the entry of the laboratory table that links to the selected laboratory by the student. If everything is fine, the server response contains the week number of the session, and the attendance status of the student, else it sends an error status code and the student is notified that the attendance code for the selected laboratory is wrong. If the laboratory code from the input is right and the attendance is successful the student receives its status consisting of the positive attendance status and the points obtained in the current laboratory session.

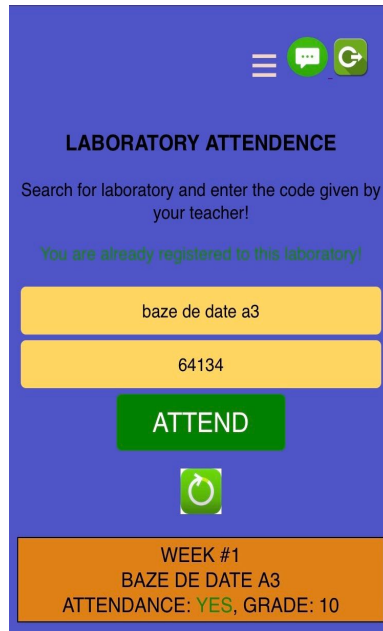


Figure 16: Screenshot representing attendance to the laboratory session page

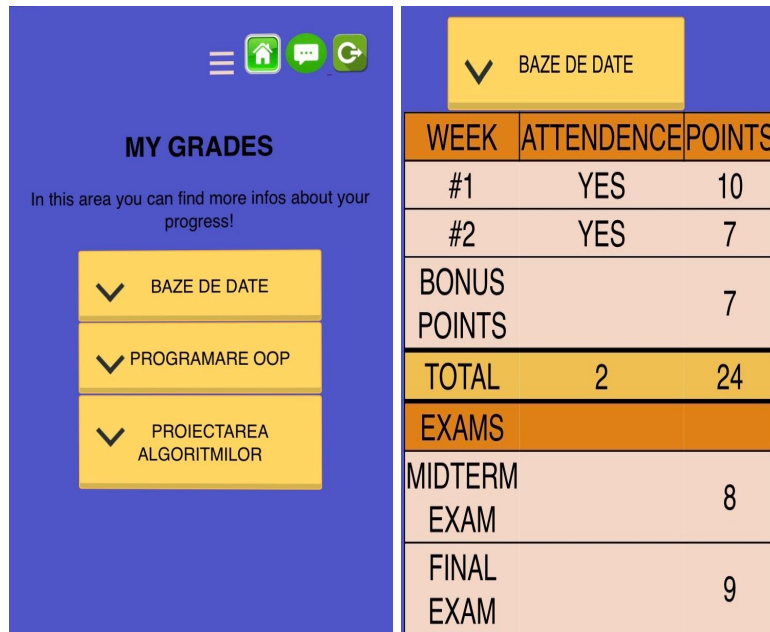
In case of successful attendance in the records table from the database, a new entry is added with the ID of the student that attended and the attendance status. When the professor grades the student, that entry will be updated with the grade and on pressing the *Refresh* button the information from the status label in the user interface will contain updated data.

The structure of the page that can be visualized in *Figure 16* and is about a simple form that wraps the input for laboratory name and represents a search bar along with all existing laboratories, the input for attendance code, the *Attend* button that produces the attendance to a certain laboratory, the *Refresh* button that has attached a function and anytime is pressed, the status of the student is updated, mostly the obtained grade. Refreshing the status label consists of rendering the data from the server response at the request made when the *Refresh* button is pressed. The label status contains the week when the current session of the laboratory took place, the full name of the laboratory and the attendance and grade status.

II.4.3 Semester overview

This page is the one that helps students to visualize their progress along the semester putting together an easily readable form of the information consisting of the number of points, attendance and grades for the midterm and final exam.

The interface elements can be visualized in *Figure 17*.



BAZE DE DATE		
WEEK	ATTENDANCE	POINTS
#1	YES	10
#2	YES	7
BONUS POINTS		7
TOTAL	2	24
EXAMS		
MIDTERM EXAM		8
FINAL EXAM		9

Figure 17: Screenshots representing semester overview of the student

The structure is made of a dynamic list of elements which includes the names of the lectures the student is registered in. Every item action a drop-down table which includes the semester point of view about the student situation on a certain lecture. This functionality is possible by uploading when the document page is ready the names and the ids of all the lectures the student is registered in, dynamically constructing a list of lecture names and attach to them an event listener function that catches the click on a certain lecture name. Once a lecture name is pressed, the server is notified to send a response to the request the information of the student registered on that lecture containing the attendances, obtained point and the exam grades if there exists any. The response has a JSON format that is parsed in javascript in order to extract all necessary data, computing the sum of the point and attendances and then construct a table below the list element of the selected lecture name and fill it with the information received.

III. Significant implementation details

III.1 Overview

The topic of this chapter is to cover some key aspects from the implementation step in the application development process and presenting a significantly used code that is similarly used in many places along with the application source code. It presents an overview of some important aspects that describe the flow of the application, how some functions act and are used, implementation details and modules used to accomplish the actual functionalities of the application.

This chapter includes important details about the implementation of the application consisting of code descriptions and significant approaches over the key code that was implemented for the application to be in the actual form and supporting the provided functionalities.

The communication between user interactions and the server-side who provides the data needed to render in the user interface is made by several requests that carry significant parameters depending on the desired action that takes place on the server and the expecting result that comes as the server response. Ajax represents the set of web development techniques that ensure the communication between client-side and server that sends data and receives data in an asynchronous way without interfering with the client interface.

The server side uses node js runtime environment to execute javascript code outside the web browser and consists of a set of endpoints that receives the post requests sent by the client and computes those requests in order to deliver the results a certain function was designed for. It includes functions that are necessary for the designed and normal flow of application behavior. The functions placed on the server-side are able to access the database and requested information required to confer security, valid and consistent data and are oriented to solve the atomic problems that have to be solved in order to facilitate functionalities provided by the application.

The area of covered problems consists of verifying if the login credentials are valid and exist in the database to authorize users to access the application. In the moment of creation of a new account, it is needed to verify if the email address was not used before by other users.

When the user wants to change its password the server has to check if the received email address is stored and linked to an existing account, generates an authentication code and sends it via

email provided by the user. The server receives that code from the user and checks if it is valid then provides authorization to the user to change the account password.

The identification of the users on the server side is based on the unique ids assigned to each user and stored in the current session to know which user wants information and for what users to provide that information. At the moment when a user having the status of a professor wants to start a session, the server verifies if there exists a created laboratory related to the user by validating the provided ID of the laboratory from the request body and of the user ID from the current session. At success, it stores the attendance code linked to the database entry of the laboratory and the week number in order to be able to manage the student information that comes when they try to attend the laboratory.

Server-side also includes the use of modules that serve functionalities needed for the application to run as the establishing connections to the services of the Google Cloud Platform derived from representative code that is used to query or to create, modify and update the data. Those modules are node js packages as *querystring* module to easily parse the request from the client, *formidable* module to easy access the profile pictures stored in the Cloud Datastore Buckets, *path* and *fs* modules meaning file system to provide easy access of the local resources in order to serve them to the clients, *express* module that stands for server configuration, *crypto* module to encrypt and decrypt the passwords of the users in the context of login validation, *express-session* module that provides the existence of a session for each user in order to store the session information as the ID of the current user, *nodemailer* module to implement the functionality of sending emails to the users in the moment of forgot password case - implementation samples provided at index [11] *Node Js Nodemailer Package* from *Bibliography* represented inspiration source for the mail system implemented in the application , *google cloud datastore* and *storage* modules which provides functions to easily access the stored data on the Cloud Platform.

III.2 Significant details

III.2.1 Connection

Connection includes the verification of user's credentials in the moment of user's authentication in order to grant their access on application. The email and password are sent to the server with a post request that has as parameters the already mentioned tuple. The following code from client-side checks if the login fields are not empty, in the affirmative case, the user is notified to complete the fields. When the given credentials are valid, the post request to the server is made.

```
/*(Client-side) Function that listen for click event on login button,
validate input, sends it to server, and treats received status code*/
$(document).ready(function() {
    $("#loginStuds").click(function() {
        var pass = $("#input_pass").val();
        var email = $("#input_email").val();
        if(pass === "" || email === "") {
            document.getElementById("status").innerHTML = "Complete all
fields!";
            return;
        }

        $.post("/check_connection",
            {email: $("#input_email").val(), pass: $("#input_pass").val()},
            function(data, status, xhr) {

                if(data === "OK"){ //email was not used by other user
                    $("#firstForm").submit();
                    return;
                }
                return;
            })
        .done(function() {return;})
        .fail(function(jqxhr, settings, ex) {
            document.getElementById("status").innerHTML = "Wrong email or
password!";
            document.getElementById("status").style.color="red";
            return;
        });
    });
});
```

The server receives the credentials and checks if valid by querying the database to check whether the email and password are assigned to any user or not. In each case the server sends as response a representative status code to help the client side to notify the user if the email or password is wrong or to redirect it to the main page of the application.

```
/*(Server-side) Endpoint that receive the post request from the client, get
parameters and validate them using checkConnection(email, pass) function*/
app.post('/check_connection', async(req,res)=>{
  let body = '';
  if (req.method === 'POST') {
    req.on('data', chunk => {
      body += chunk.toString();
    });
    req.on('end', async () => {
      var vars = parse(body);
      var email = vars.email;
      var pass = vars.pass;
      try{
        var exists = await model.checkConnection(email,pass);
        if(exists){
          res.status(200).send('OK');
        }
        else{
          res.status(406).send('Conflict');
        }
      }
      catch(e){
        console.log(e);
      }
    });
  }
});

//Function that validate email and password
var checkConnection = async(email, pass) =>{
  pass = encodePass(pass);
  var query = datastore.createQuery('login').filter('email', '=',
email).filter('pass', '=',pass);
  var [tasks] = await datastore.runQuery(query);

  if(tasks[0]== undefined)return false;
  return true;
}

//Function that encodes the password
var encodePass = pwd => crypto.createHash('sha256').update(pwd).digest('hex');
```

III.2.2 Messenger Service

In the chapter *Application structure elements and functionalities* was mentioned the fact that messenger is composed from three parts: the front messenger that includes the list of persons a user communicate with, the conversation between current user and another person and the search part of the messenger where a user can search a people to text with.

The following code computed on client-side loads the people the current user has a conversation with using the data that came in server response. The persons are sorted by priority of unread messages. Then performs populating the front list with persons calling *constructFront()* function.

```
/*(Client-side) Function that loads persons from the front list*/
var people = []
var currentSelected;
var loaded = false;
async function loadPersons(){
    loaded = false;
    people = [];
    await $.post("/load_persons_for_messenger")
    .done(async function(persons) {
        if(!persons) return;
        //add firstly the people with unread messages by the user
        for(let i in persons){
            if(persons[i].seen == false)
                people.push(persons[i])
        }
        //add secondly the people with read messages by the user
        for(let i in persons){
            if(persons[i].seen == true)
                people.push(persons[i])
        }
        var namesBox = document.getElementById("names");
        namesBox.innerHTML = "" //persons's list is made empty
        loaded=true;
        await constructFront()
    });
}
```

```
/*(Server-side) Endpoint that receive the request to load user's contacts*/
app.post('/load_persons_for_messenger', async(req,res)=>{
```

```

        var persons = await model.getPersons(req.session.user_id);
        res.json(persons);
    });

    /*Function that receives the user id and query database for the conversation
    it has with other people*/
    var getPersons = async (user_id)=>{
        var query = datastore.createQuery('messenger').filter('to_id', '=',
user_id);
        var [pers] = await datastore.runQuery(query);
        var queryi = datastore.createQuery('messenger').filter('from_id', '=',
user_id);
        var [pers2] = await datastore.runQuery(queryi);

        var sorted_persons =[]
        if(pers!=undefined)
            for(let i in pers){
                sorted_persons.push(pers[i])
                //console.log("pers", pers[i])
            }
        if(pers2!=undefined)
            for(let i in pers2){
                sorted_persons.push(pers2[i])
                //console.log("pers2", pers[i])
            }
        //sort by date persons who sent messages
        var temp = []
        var ids = []
        if(sorted_persons!=undefined){
            sorted_persons = sorted_persons.sort((a, b) => a.time - b.time)
            //get ids of user_id contacts
            for(let i in sorted_persons){
                if(sorted_persons[i].from_id == user_id)
                    temp.push(sorted_persons[i].to_id)
                if(sorted_persons[i].to_id == user_id)
                    temp.push(sorted_persons[i].from_id)
            }
            //get unique ids of user_id contacts
            ids = Array.from(new Set(temp));
        }

        var p_buff =[]
        if(ids!=undefined){
            //for every contact get the messages and add to json array p_buff along with
            seen status
            for(let i in ids){
                var p_id = ids[i];
                var q = datastore.createQuery('messenger')
                    .filter('text', '=', p_id+user_id)
                var [unread] = await datastore.runQuery(q);
                var seen = true;
                if(unread[0]!=undefined)
                    seen = unread[0].seen;
            }
        }
    }

```

```

        const [entity] = await datastore.get(datastore.key(['login',
parseInt(p_id)]));
        var url = await getMyProfileUrl(entity[datastore.KEY].id)
        p_buff.push({name:entity.firstname+" "+entity.lastname,
id:entity[datastore.KEY].id, seen: seen, profile_url:url})
    }
}
return p_buff;
}

```

The `constructFront()` function also attaches an on click event listener `sendMsg()` to the *send message* button to catch when the user sends a new message to another user. The `sendMsg()` function gets the message, validates to not be empty then sends it to the server-side along with the user id that has to receive the message. When the server operation is done, meaning the message was recorded in the database, the new message is added to the current conversation. The text box that carries the message input is made clear.

```

/*(Client-side) Click event function that gets the message from message box
and sends it to the server in order to be stored, then reload the current
conversation*/
async function sendMsg() {
    var input = document.getElementById("textMsg")
    var text = input.value
    if(text == "")return
    var sendToId;
    if(fromSearch)
        sendToId = cSpeaker
    else
        sendToId = cSpeaker
    await $.post("/send_message_to",{id:sendToId, text:text})
        .done(async function() {
            await loadConversation(sendToId)
            input.value = "";
        });
}

```

```

/*(Server-side) Endpoint that receives the request to store a message from
user and a certain destination user id*/
app.post('/send_message_to', async (req, res) => {
    let body="";
    req.on('data', chunk => {
        body += chunk.toString();
    });
    req.on('end', async () => {
        var vars = parse(body);
    });
}

```

```

        await model.sendMessage(vars.id, vars.text, req.session.user_id);
        //console.log("tasks"+tasks);
        res.status(200)
        res.send()
    });
});
/*Function that store the new message send by from_id user to to_id user*/
var sendMessage = async(to_id, text, from_id)=>{
    var time = new Date();
    var seen = false;
    await datastore.save({
        key: datastore.key('messenger'),
        data: {text, from_id, to_id, time, seen}})

    var query = datastore.createQuery('messenger').filter('text', '=',
from_id+to_id);
    var [status] = await datastore.runQuery(query);
    if(status[0]!==undefined){
        const [entity] = await datastore.get(status[0][datastore.KEY])
        entity.seen = false
        await datastore.update(entity)
    }
    else{
        text = from_id + to_id;
        from_id = ""
        to_id = ""
        await datastore.upsert({
            key: datastore.key('messenger'),
            data: {text, from_id, to_id, time, seen}})
    }
}
}

```

Every person shown in the front messenger has attached an on click event listener that opens the conversation between the current user and the pressed person. A post request is sent to the server in order the client-side to receive the data needed to build the conversation. Once messages are loaded then the *loadConversation()* function is called in order to correctly render the received data.

```

/*(Client-side) Function that load conversation between current user and
another user identified by p_id, renders the received data*/
async function loadConversation(p_id){
    await $.post("/load_conversation",{id:p_id})
        .done(convs => {
            if(!convs) return;
            var convsBox = document.getElementById("conversation")
            convsBox.innerHTML = ""

```

```

        for(let i in convs){
            var p = document.createElement("DIV")
            var span = document.createElement("SPAN")
            span.className = "messageFragment"
            span.innerHTML = convs[i].text
            p.appendChild(span)
            if(convs[i].position == "left"){
                p.className = "left"
            }
            else{
                p.className = "right"
            }
            convsBox.appendChild(p)
        }
    });
    var currentConversation = document.getElementById("conversation");
    currentConversation.scrollTop = 10000000;
}

```

```

/*(Server-side) Endpoint that receives the request to load and send
conversation between two users identified by their ids*/
app.post('/load_conversation', async (req, res) => {
    let body="";
    req.on('data', chunk => {
        body += chunk.toString();
    });
    req.on('end', async () => {
        var vars = parse(body);
        var messages = await model.loadConversation(vars.id,
req.session.user_id);
        res.json(messages)
    });
});

/*Function that query database for all the messages between two users, */
var loadConversation = async (req_id, current_id)=>{
    var query = datastore.createQuery('messenger').filter('to_id', '=',
current_id).filter('from_id', '=', req_id);
    var [pers] = await datastore.runQuery(query);
    var queryi = datastore.createQuery('messenger').filter('from_id', '=',
current_id).filter('to_id', '=', req_id);
    var [pers2] = await datastore.runQuery(queryi);

    var kk = req_id+current_id
    var q = datastore.createQuery('messenger')
        .filter('text', '=', kk)
    var [read] = await datastore.runQuery(q);
    if(read[0]!==undefined){
        const [status_row] = await datastore.get(read[0][datastore.KEY]);

```

```

        status_row.seen = true;
        await datastore.update(status_row)
    }
    var sorted_persons = []
    if(pers!=undefined)
        for(let i in pers){
            sorted_persons.push(pers[i])
        }
    if(pers2!=undefined)
        for(let i in pers2){
            sorted_persons.push(pers2[i])
            // console.log("pers2", pers[i])
        }
    //sort by date the sent messages
    var temp = []
    var left = "left"
    var right = "right"
    if(sorted_persons!=undefined){
        sorted_persons = sorted_persons.sort((a, b) => a.time - b.time)
        for(let i in sorted_persons){
            if(sorted_persons[i].from_id == current_id)
                temp.push({text:sorted_persons[i].text, position: right})
            if(sorted_persons[i].to_id == current_id)
                temp.push({text:sorted_persons[i].text, position: left})
        }
    }

    return temp;
}

```

Messenger service also provides a notification system that notifies users every time they receive a new message, on every page the surf. This functionality is accomplished by cyclic requests to the server in order to check if there are new incoming messages. In the affirmative case the user is notified. The script that computes this behavior is presented in the following code sample. Once the data is received the focus is on unread messages that are counted and notification number is shown in the user interface.

```

/*(Client-side)*/
//Notify when new messages appear
async function loadPersons(){
    await $.post("/load_persons_for_messenger")
        .done(persons => {
            if(!persons) return;
            var count = 0
            for(let i in persons){
                if(persons[i].seen ==false)

```



```

        count += 1
    }
    if(count != 0){
        document.getElementById("countSeen").style.display = "block"
        document.getElementById("countSeen").innerHTML = count
    }

    });
}
setInterval(() => {loadPersons();}, 6000)
loadPersons()

```

The server function that loads the messages along with the seen status of every message received by the user has been presented above.

III.2.3 Create a new lecture or laboratory

The pages that offer the functionality of creating new laboratories or lectures send a request to the server containing the professor ID and the input data from the forms in order to store it in the database. Every action of this kind before being computed the server validates the user request in order to not have duplicated data where the design was to have unique stored data.

On the client-side when the *Create* button is pressed, the input data (name, acronym) is validated, then a request is sent with this data along with automatically generated lecture code, to the server that has to perform the action of creating a new laboratory or lecture. On success or in case of an already existing lecture, the user is notified.

```

/*(Client-side) Function that validate input data from the create new
lecture form, send request to the server to create a new lecture assigned to
the current user identified by its id*/
async function createLecture(){
    labname = $("#labName").val().toUpperCase();
    if(labname == ""){
        document.getElementById("respond").innerHTML = "Please enter the name of
laboratory!";
        document.getElementById("respond").style.color = "red";
        return;
    }
    acronym = $("#lectureAcronym").val().toUpperCase();

```

```

    if(acronym == ""){
        document.getElementById("respond").innerHTML = "Please enter the
acronym!";
        document.getElementById("respond").style.color = "red";
        return;
    }
    code = document.getElementById("lectureCode").value;
    $.post("/create_lecture",
        { myData: labname, acro: acronym, code:code },
        async function(data, status, xhr) {
            console.log("status at return"+data+status+xhr);

            if(status==="success"){
                document.getElementById("lectureCode").value = code;
                document.getElementById("respond").innerHTML = "Lecture created!
Provide this generated code needed for laboratory creation!";
                document.getElementById("respond").style.color = "green";

            }
            else if(status ==="cant"){
                document.getElementById("respond").innerHTML = "Can't create...";
                document.getElementById("respond").style.color = "red";
            }
        })
    .done(function() { /* alert('Request done!'); */ })
    .fail(function(jqxhr, settings, ex) {
        document.getElementById("respond").innerHTML = "Can't create,
this lecture already exists!";
        document.getElementById("respond").style.color = "red"; }
    );
}

```

```

/*(Server-side) Endpoint that receive the data of the lecture that has to be
created*/
app.post('/create_lecture', async(req,res)=>{
    let body="";
    req.on('data', chunk => {
        body += chunk.toString();
    });
    req.on('end', async () => {
        var vars = parse(body);
        var result = await model.createLecture(vars.myData, vars.acro,
vars.code, req.session.user_id);
        if(result){res.json({success: 'Register successfully.'});}
        if(result == false){ res.status(406).send('Already exists');}
    });
});

/*Function that validate the data corresponding to the new lecture that will
be created and create it if valid data was provided*/
var createLecture = async (lname, acro, code, prof_id)=>{

```

```

    var result;
    //check if lecture code was not used yet, it is primary key
    const queryi = datastore.createQuery('lectures').filter('lecture_code',
    '=', code.toString())
    .limit(1);
    await datastore
    .runQuery(queryi)
    .then(data =>{
        result = data;
    });
    if(result[0].length > 0)
        return false;
    //check if new lecture was not created before
    const query = datastore.createQuery('lectures').filter('acronym', '=',
    acro.toString())
    .filter('name', '=', lname.toString())
    .limit(1);
    await datastore
    .runQuery(query)
    .then(data =>{
        result = data;
    });
    console.log(result[0]);
    if(result[0].length > 0)
        return false;
    //initialize needed values to be stored at lecture creation
    var registered;
    var acronym = acro;
    var labs_code = "";
    var lecture_code = code;
    var name = lname;
    var owner_id = prof_id;
    //Create a new lecture
    const reg = await datastore.save({
    key: datastore.key('lectures'),
    data: {acronym, labs_code, lecture_code, name, owner_id}})
    .then(function onSuccess(entity){
        registered = true;
    })
    .catch(function onError(error){
        registered = false;
    });
    if(registered)
        return true;
    return false;
}

```

The server receives the name, acronym, and generated code, along with the saved on current session user id, validates the data and creates the new lecture assigned to a certain professor identified by the user id.

The creation of a new laboratory is quite the same, except that the acronym is not needed and at the moment of creating a new laboratory the user has to know the lecture code provided by the lecture owner.

III.2.4 Laboratory sessions

On starting laboratory sessions, after the professor completes the session details as target laboratory, attendance code and the week when the attendance session takes place, starts the cyclic process of updating students's list in order to provide real-time connection of the students to the session. The data that is continuously updated are the student names, their personal information and the points if already given by the professor. The load process of data consists of a request to the server that identifies the professor by the saved user id in the current session and query the database for the data that rely that students attended to his attendance session, meaning new entries in the Records table along with the id of the professor.

The most significant implementation details consists of loading the profile picture for each student from the student's list, the mechanism that takes place when the professor grade a certain student, when choose to eliminate a student from the attendance session, when search for a student in the student's list or at the moment when the professor request more personal information about a student in order to better identify his person.

For loading the profile picture of every student, on the client-side is sent a request to the server along with the student id it wants the profile picture url for.

```
/*(Server-side) Endpoint that receive the request of query the Google  
Storage Bucket in order to send to the client-side the url of the profile  
image to a certain student identified by its user id*/  
var getStudentsUrl = async (user_id) => {  
  console.log("INDEX primesc id pt test url", user_id)  
  var img_name = user_id + '.png'  
  var url;  
  var file = await storage.bucket(BUCKET_NAME).file(img_name)  
  let promise = new Promise(async function(resolve, reject) {  
    await file.getMetadata(function(err) {  
      if(err!=null){  
        if (err.code === 404) {  
          resolve("no")  
        }  
      }  
    }  
    else{  
      url =
```

```

`https://storage.googleapis.com/${BUCKET_NAME}/${img_name}`
    resolve(url)
  }
});
})
await promise.then(
  result => {
    if(result == "no")
      url = undefined
    else
      url = result
  });
if(url != undefined){
  return url;
}
else{
  return undefined;
}
}
}

```

When a professor wants more personal information about a certain student, on client-side is sent a request with the student id the information is requested for.

```

/*(Client-side) Function that send request to the server for student informations*/
async function getGroupAndMatricol(stud_id){
  try{
    await $.post("/load_students_infos_session",
      { student_id: stud_id})
    .done(infos => {
      console.log("grupa"+infos[0]+infos[1]);
      var group = document.getElementById("groupOfSelectedStudent");
      group.innerHTML = "GRUPA: <br>"+infos[1].toUpperCase();
      var id = document.getElementById("idOfSelectedStudent");
      id.innerHTML = "NR MATRICOL:<br>"+infos[0].toUpperCase();
    })
    .fail(function(){
      var group = document.getElementById("groupOfSelectedStudent");
      group.innerHTML = "GRUPA: <br>"+"NOT SET";
      var id = document.getElementById("idOfSelectedStudent");
      id.innerHTML = "NR MATRICOL:<br>"+"NOT SET";
    });
  }
}

```

```

/*(Server-side) Function that receive the request for get the student information identified by the student id, the id is sent to the function that query the database in order to obtain those information and send them

```

```

back to the client in order to be displayed after the professor request*/
app.post('/load_students_infos_session', async (req, res) => {
  let body="";
  req.on('data', chunk => {
    body += chunk.toString();
  });
  req.on('end', async () => {
    var vars = parse(body);
    var infos = await model.getStudentsInformations(vars.student_id);
    res.json(infos);
  });
});

var getStudentsInformations = async (student_id)=>{
  var query = datastore.createQuery('student_infos').filter('student_id',
  '=', student_id);
  var [tasks] = await datastore.runQuery(query);
  var infos = []
  if(tasks[0]!==undefined){
    infos[0] = tasks[0].nr_matricol;
    infos[1] = tasks[0].group;
    return infos;
  }
  return undefined;
}

```

When the professor chooses to search a student in the search bar over the student's list, is called a javascript function that fetches the table's rows that compose the student's list by the input search criteria.

```

/*(Client-side) Function that fetches the table's rows containing the students that are registered to the current attendance session*/
function searchStudent() {
  var input, filter, table, tr, td, i, txtValue;
  input = document.getElementById("searchStudentInput");
  filter = input.value.toUpperCase();
  table = document.getElementById("studentsTable");
  tr = table.getElementsByTagName("tr");
  for (i = 1; i < tr.length; i++) {
    td = tr[i].getElementsByTagName("td")[0];
    if (td) {
      txtValue = td.textContent || td.innerText;
      if (txtValue.toUpperCase().indexOf(filter) > -1) {

        tr[i].style.display = "";
      } else {
        tr[i].style.display = "none";
      }
    }
  }
}

```

```
}  
}
```

A professor that starts an attendance session has the capability to grade students. When the professor wants to grade a student, the Grade button is pressed. This button has an on click event listener function attached to it that sends a request to the server to grade the student along with the grade and with other data needed to record the grade related to the student. Every line in the student's list has an id that is taken in the process of grading and with the help of it is identified the student id that has to be graded.

```
/*(Client-side)Function that send the data of the student to the server that  
compute the operation of recording the points obtained by a student  
identified by user id*/  
gradeBtn.onclick=function() {  
    var inp=document.getElementById("i"+this.id.substr(1,this.id.length));  
    var idd = this.id;  
    if(inp.value!=""){  
        pseudoGrades[i] = inp.value;  
document.getElementById("studentsTable").rows[parseInt(this.id.substr(1,this.i  
d.length),10)+1].cells[1].innerHTML = inp.value;  
        var id=studentIds[parseInt(this.id.substr(1,this.id.length),10)];  
        $.post("/update_grade",{ myLab: currentLabName, myGrade: inp.value,  
myId: id, week: currentWeek, lec_code: lectureCode })  
        .done(function() {  
            pseudoGrades[i]=studentGrades[i];  
        });  
        inp.value="";  
    }  
}
```

```
/*(Server-side) Function that receives the laboratory name, the grade,  
student id, week number, lecture code and professor id in order to grade a  
student*/  
app.post('/update_grade', async (req, res) => {  
    let body="";  
    req.on('data', chunk => {  
        body += chunk.toString();  
    });  
    req.on('end', async () => {  
        var vars = parse(body);  
        var tasks = await model.updateGrade(vars.myLab, vars.myGrade,  
vars.myId, vars.week, vars.lec_code, req.session.user_id);  
        //console.log("tasks"+tasks);  
        res.status(200).send('grade was updated');  
    });  
});
```

```

});

/*Function that receives the data needed to record a student grade in the
database, using id is obtained the week of the laboratory and the key needed
to be stored in the Records table along with the student informations*/
var updateGrade = async(lab_name, grade, stud_id, week_no, lecture_code,
prof_id) =>{
    //get lab_id
    var query = datastore.createQuery('labs').filter('name', '=',
lab_name).filter('prof_id', '=', prof_id);
    var [tasks] = await datastore.runQuery(query)
    //get the key of the laboratory with the id given as parameter
    var lab_id = tasks[0][datastore.KEY].id;
    var week_number = week_no; //tasks[0].week_number;
    // get students ids on this lab at this week
    var queryi = datastore.createQuery('records').filter('lab_id', '=',
lab_id).filter('week_number', '=', week_number).filter('student_id', '=', stud_id)
;
    var [students] = await datastore.runQuery(queryi);
    const k = students[0][datastore.KEY];
    const [entity] = await datastore.get(k);
    entity.grade=grade;
    entity.lecture_code = lecture_code;
    await datastore.update(entity);
}

```

Professor has also the ability of remove a student form the current laboratory session, this is done by pressing the *Remove* button from the popup box containing personal information about a student, has attached a function that request the server to remove the student by providing to it laboratory identification details and the id of the student that has to be removed.

```

/*(Client-side) Function attached to Remove button that performs request to
the server to remove a certain student, once done the student is eliminated
and the student's list is updated*/
deleteStudent.addEventListener('click', async function(){
    if(removePressed == false){
        await $.post("/removeStudent",
            { myLab: currentLabName, student_id: studentIds[index]})
        .done(async function(result) {
            var status = document.getElementById("statusPopup");
            if(result){
                removePressed = true;
                status.innerHTML="Student will be removed.<br>Updates will
appear soon!";
                status.style.color = "yellow";
            }
            else{
                status.innerHTML="Cannot remove!";
            }
        })
    }
})

```



```

        status.style.color = "red";
    }

    });
}
});

```

On the server-side, the function that treats this request, receives needed data and perform the action of student removal.

```

/*(Server-side) Function that treats the request of student removal. The parameters from request are extracted and sent to the function that performs student removal action*/
app.post('/removeStudent', async (req, res) => {
    let body="";
    req.on('data', chunk => {
        body += chunk.toString();
    });
    req.on('end', async () => {
        var vars = parse(body);
        var respond = await model.removeStudent(vars.myLab, vars.student_id, req.session.user_id);
        //console.log("tasks"+tasks);
        res.send(respond);
    });
});

/*Function that performs student removal identified by id from a certain laboratory. Function checks if the laboratory is valid in order to get its key to perform a query in the Records table to remove the student from the given laboratory session of the professor*/
var removeStudent = async(lab_name, student_id, prof_id)=>{
    //get lab_id
    var query = datastore.createQuery('labs').filter('name', '=', lab_name).filter('prof_id', '=', prof_id);
    var [tasks] = await datastore.runQuery(query)
    var lab_id = tasks[0][datastore.KEY].id;
    var week_number = tasks[0].week_number;

    var deleted;
    // get students ids on this lab at this week
    var queryi = datastore.createQuery('records').filter('lab_id', '=', lab_id).filter('week_number', '=', week_number).filter('student_id', '=', student_id);
    var [student] = await datastore.runQuery(queryi);
    var key = student[0][datastore.KEY];
    await datastore.delete(key)
    .then(function onSuccess(entity){
        deleted = true;
    });
}

```

```
    })  
    .catch(function onError(error) {  
        deleted = false;  
        console.log("error register"+error);  
    });  
    return deleted;  
}
```

In order to accomplish the functionality of real-time flow data on the application, it was necessary to find a way to cyclic request the server for the needed information. Situations where this behavior was necessary to represent the laboratory session page to continuously update the students that attend and at the messenger service to notify the users of incoming messages. For this functionality to take place the solution was to cyclic call the functions scoped to request the server and get the required sensitive information. The javascript provides the method of setting intervals to accomplish this behavior. The following lines represent the implementation of this aspect. For example, when the laboratory session is started the information about students is loaded continuously to update the student list in order for the professor to see the users that attend his laboratory. When the *Stop* button is pressed then there is no need for the student's data to be updated, therefore the interval from time to time the function scoped to update the student list is removed.

```
var interval = setInterval(load_students, 60000);  
clearInterval(interval);
```

III.2.5 Stored data overview

Generally, an overview of data consists of visualising stored data relative to a laboratory, lecture, or the personal situation of a student. The tables that are dynamically built in the user interface use data sent by the server after a request from the client-side.

III.2.5.1 Professors overviews

Professors overview are met on the process of managing stored data about students that attended a certain laboratory on a certain week, or along the whole semester. When the professor wants to visualize recorded data relative to a certain laboratory, on the client-side is sent an request to the server containing the laboratory id and with the user id stored in the current session, the server is able to send to the client-side needed data to build overviews in the user interface.

In the following code sample, the request to the server is sent in order to receive the data(student's names, ids and grades) for visualizing the students that attended a laboratory on a certain week. Once the requested data is received, the function dynamically built the visualization table that composes the overview.

```
/*(Client-side) Function that performs server request in order to get the data
about student's situation on a laboratory, on a certain week*/
async function loadStudentsNames(week_no){
    await $.post("/load_names_per_week",
        { myLab: currentLabName, week_no: week_no})
    .done(labs_list => {
        if(!labs_list) return;
        currentStudents = [];
        currentStudents = currentStudents.concat(labs_list);
        console.log(currentStudents);
    });

    await $.post("/load_students_ids_per_week",
        { myLab: currentLabName, week_no: week_no})
    .done(labs_list => {
        if(!labs_list) return;
        studentIds = [];
        studentIds = studentIds.concat(labs_list);
        console.log(studentIds);
    });
};
```

```

    await $.post("/load_students_grades_per_week",
    { myLab: currentLabName, week_no: week_no})
    .done(labs_list => {
        if(!labs_list) return;
        studentGrades = [];
        studentGrades = studentGrades.concat(labs_list);
        console.log(studentGrades);
    });
}

```

Another example of overview is when the professor wants to visualize the situation of every student from the semester point of view. The request is made to receive needed data for a certain lecture identified by lecture code.

```

/*(Client-side)Function that request students data as name, personal infos and
semester situation*/
async function loadStudents(lab_name){
    await $.post("/load_students_profile_per_semester",
    { myLab: lab_name})
    .done(students_list => {
        if(!students_list) return;

        for(let i=0; i<students_list.length;i++){
            currentStudents[i] = students_list[i];
            console.log(students_list[i].firstname + " "+
students_list[i].lastname );
            console.log(students_list[i]);
        }
    });
}

```

On the server-side, when a request like this one is received, the data from parameters is sent to the function that handles this type of request, providing a semester situation of the students registered on a certain lecture. Using the laboratory id provided at server request, the students registered along the semester at that laboratory are obtained and with the help of their ids and data saved on the *Records* table, the situation is sent to the client as a json that contains an element with the profile informations, grades and attendances on each week out of fourteen, for each student. The students are alphabetically sorted in order to provide the client-side easy readable data for the professors.

```

/*(Server-side) Function that receives the request and sends data received to
the function that provides the semester situation of students*/

```

```

app.post('/load_students_profile_per_semester', async (req, res) => {
  let body="";
  req.on('data', chunk => {
    body += chunk.toString();
  });
  req.on('end', async () => {
    var vars = parse(body);
    var tasks = await model.getStudentsProfilePerSemester(vars.myLab,
req.session.user_id);
    res.json(tasks);
  });
});

var getStudentsProfilePerSemester = async(lab_name, prof_id)=>{

  //get lab_id
  var query = datastore.createQuery('labs').filter('name', '=',
lab_name).filter('prof_id', '=', prof_id);
  var [tasks] = await datastore.runQuery(query)
  var lab_id = tasks[0][datastore.KEY].id;

  // get all students on this semester at this object
  var queryi = datastore.createQuery('records').filter('lab_id', '=',
lab_id);
  var [students] = await datastore.runQuery(queryi);
  var students_ids=[];
  //get all students ids
  for(let i in students) {
    students_ids[i]=students[i].student_id;
  }

  // get unique students ids
  const ids = Array.from(new Set(students_ids));
  students_profile = []

  var nr_matricol;
  var group;
  for(let j in ids){
    //get student nr matricol and group
    var infos=[];
    nr_matricol = "";
    group = "";
    infos = await getStudentsInformations(ids[j]);
    if(infos != undefined){
      if(infos[0] != undefined) nr_matricol = infos[0];
      else nr_matricol = "-";
      if(infos[1] != undefined) group = infos[1];
      else group = "-";
    }

    //get student firstname, lastname
    const [student] = await datastore.get(datastore.key(['login',

```

```

parseInt(ids[j]))));
    if(student!=undefined){
        //get student grades and attendences
        var student_situation = await
getLaboratoryInformationsForName2(lab_name, ids[j]);
        var
w1="none",w2="none",w3="none",w4="none",w5="none",w6="none",w7="none",w8="none
",w9="none",w10="none",w11="none",w12="none",w13="none",w14="none";
        for(let k=0; k<student_situation.length; k++){

            if(student_situation[k].week_number!= undefined){
                switch(student_situation[k].week_number.toString()){
                    case "1": w1 = student_situation[k].grade; break;
                    case '2': w2 = student_situation[k].grade; break;
                    case '3': w3 = student_situation[k].grade; break;
                    case '4': w4 = student_situation[k].grade; break;
                    case '5': w5 = student_situation[k].grade; break;
                    case '6': w6 = student_situation[k].grade; break;
                    case '7': w7 = student_situation[k].grade; break;
                    case '8': w8 = student_situation[k].grade; break;
                    case '9': w9 = student_situation[k].grade; break;
                    case '10': w10 = student_situation[k].grade; break;
                    case '11': w11 = student_situation[k].grade; break;
                    case '12': w12 = student_situation[k].grade; break;
                    case '13': w13 = student_situation[k].grade; break;
                    case '14': w14 = student_situation[k].grade; break;

                }
            }
        }

        students_profile.push({id: ids[j], firstname: student.firstname,
lastname: student.lastname, nr_matricol: nr_matricol, group: group, w1: w1,
w2: w2, w3: w3, w4: w4, w5: w5, w6: w6, w7: w7, w8: w8, w9: w9, w10: w10, w11:
w11, w12: w12, w13: w13, w14: w14 });
    }
}
//sort students alfabetically
for(var i=0;i<students_profile.length-1;i++){
    var first =
students_profile[i].firstname.toLowerCase().charCodeAt(0);
    for(var j =i+1; j<students_profile.length; j++){
        var second =
students_profile[j].firstname.toLowerCase().charCodeAt(0);
        console.log("s"+second);
        if(first >second){
            //swap students names
            var aux = students_profile[i];
            students_profile[i] = students_profile[j];
            students_profile[j] = aux;
            first =
students_profile[i].firstname.toLowerCase().charCodeAt(0);

```

```

        }
    }
}
return students_profile;
}

```

III.2.5.2 Students overview

The overview of each student situation that is accessed in the history page of the student side is constructed by referring to the ID of the student. The database is queried to get the grades on each laboratory the student attended identified by a certain id.

The profile configuration on the student side consists of the personal information of a student identification in real life that is stored in the Cloud Storage Platform.

Before getting the semester situation on a certain lecture, those lectures the student is registered at are loaded, an event listener is set for every one that at the click event, the client-side sends a request to the server to serve the semester overview of a certain student on a certain lecture. The student id is stored in the current session, only the lecture code that was previously loaded is provided to the server in order to accomplish the task.

On the following sample is presented the server flow when receiving this kind of request.

```

/*(Server-side) The lecture code is provided in order to receive the student
situation along the semester*/
app.post('/load_laboratory_infos_for_name', async (req, res) => {
    let body="";
    req.on('data', chunk => {
        body += chunk.toString();
    });
    req.on('end', async () => {
        var vars = parse(body);
        var grades = await model.getLectureGradesForStudent(vars.lecture_code,
req.session.user_id);
        res.json(grades);
    });
});
/*Function that query the database in order to get all the student's grades
obtained along the whole semester. Those grades are sorted in order to
respect the logical flow of the laboratories.*/

```

```

var getLectureGradesForStudent = async (lecture_code, student_id) => {
    var query =
datastore.createQuery('records').filter('student_id', '=', student_id).filter('lecture_code', '=', lecture_code);
    var [student_rows] = await datastore.runQuery(query);
    var infos = [];
    var week = 0;
    var index = 0;
    for (week = 1; week <= 14; week++) {
        for (let j in student_rows) {
            if (week === parseInt(student_rows[j].week_number, 10)) {
                infos[index] = student_rows[j];
                index = index + 1;
                break;
            }
        }
    }
    return infos;
}

```

III.2.6 General implementations samples

The pages that require searching or selecting a laboratory from a list are based on the loaded information from the server when the document page is ready and accessible to the user. The information as the name of the laboratories and their ids are loaded in order for the client to be able to send valid requests to the server from the client-side and manage the provided data along with the user actions. The search bar that offers to select an element, navigation over the result with the up and down arrow keys and provides on key up suggestions were implemented by inspiration of an example found on the internet at the web address.

The link found at index [9] *Input Autocomplete Functionality* from *Bibliography* contains a code sample of how to implement this functionality and represents a source of inspiration but the code was modified to serve the personal use of the application.

The pages that contain visualizations of students list before being accessible to the user that information was loaded such as names of the students, their personal information for identification, their grades and attendance number along with each ID of the student. When a

user performs an action on a certain student the received request on the server contains in the body the ID of the student and using the ID of the professor store in the current session, those actions are well computed to accomplish the desired application flow to store new data or to receive data by maintaining the persistence of the untargeted stored data.

On the following lines is described as a sample of code that consists of a request from the client-side to the server-side to get the code of a lecture the laboratory is related to by providing the LaboratoryID that was previously loaded on the client-side. The function call is asynchronous to wait for the server to complete the request and to return the requested data as a response to the client. The post request is made with ajax mentioned above and receives as parameters the URL endpoint where the server listens for this kind of action and the LaboratoryID that represents the body of the request as JSON format. When the request is done the lecture code is used on the client-side to identify the lecture that owes the selected laboratory.

```
async function getLectureId(laboratory_id){
  await $.post("/get_lecture_code",{lab_id:laboratory_id})
    .done(lecture_code => {
      if(!lecture_code) return;
      lectureCode = lecture_code.code;
    });
}
```

The server receives the request from the client as a post method, parses the request in order to get the parameters from the request body needed to perform the action the function is designed to and sends those parameters to the model module that contains the functions to access the stored data from the cloud. It is an asynchronous call because the function has to await the returned data from the *getLabCodeForLecture()* function in order to send it as a response to the client.

```
app.post('/get_lecture_code', async (req, res) => {
  let body="";
  req.on('data', chunk => {
    body += chunk.toString();
  });
  req.on('end', async () => {
```

```

        var vars = parse(body);
        var code = await model.getLabCodeForLecture(vars.lab_id);
        res.json({code:code});
    });
});

```

On the module side, the function takes the laboratoryID and performs a query on the datastore where the data is stored in the Cloud Platform. This query is provided by the datastore object that manages the connection and operations needed for accessing data. When the query is done the functions return the lecture code for the received laboratoryID as a parameter.

```

var getLabCodeForLecture = async (laboratory_id) => {
    Var query =
    datastore.createQuery('lectures').filter('labs_code','=',laboratory_id);
    var [labs] = await datastore.runQuery(query);
    return labs[0].lecture_code;
}

```

The link at index [12] *Node Js Promises* from *Bibliography* contains tutorials about the usability of promises that were an inspiration source and the studied examples were modified to meet the application functionalities. At the moment when the application deals with asynchronous functions in order to await the response from the made requests, it is needed to use promises that help to accomplish this task.

In order to provide the functionality of restoring some actions and behaviors of the application, cookies were used for storing essential key parameters to rebuild the user interaction that could offer usability to the users. For example, if a professor starts a laboratory session and leaves application, at the moment he comes back the started session will be restored unless it has been stopped.

The following link found at index [2] *Cookies Overview* from *Bibliography* contains code samples that use cookies that represent an inspiration source for the implementation of some functionalities of the application. The code was modified and integrated to serve the application functions.

The structure of the menu that links the Html pages was inspired by the sample code found at index [13] *Side Navigation Menu* from *Bibliography* that was translated and modified to meet the design of the application. The sample code presented below provides the

implementation of showing and hiding the menu when the user clicks the *Menu* button in order for it to appear if it is not visible to the user and to disappear if it is shown to the user.

In the following script a global variable that store if the menu is shown or not is used and the *openNav()* function that is assigned to the *Menu* button checks the cases of boolean value of the *openMenu* variable depending on its value of true or false, the properties of the menu are modified in order for the menu element to be or not be shown on the user interface.

```
var openMenu = false;
function openNav() {
    if(!openMenu){
        document.getElementById("mySidenav").style.width = "250px";
        document.getElementById("main").style.marginLeft = "250px";
        openMenu = true;
    }
    else{
        closeNav();
        openMenu = false;
    }
}
function closeNav(){
    openMenu = false;
    document.getElementById("mySidenav").style.width = "0";
    document.getElementById("main").style.marginLeft= "0";
}
```

Conclusions

The Laboratory Management Platform is easy to use and access application meant to facilitate the grading system and attendances during the semester, offering a series of functionalities that give to the application a pleasant experience as well as a form that satisfies the needs of the approached topic being well elaborated. Provides secure access to users on the platform and provides integrity and security of data stored in the Google Cloud. It is a beneficial method in the field of education, being a work that took shape after many hours of documentation and involvement and is a better variant of student situation management than the classic one on sheets of paper.

The thesis is well structured due to the form it has and includes relevant information that exposes a detailed perspective on the developed application related to architecture, functionality and implementation details. It includes an easy to understand language with screenshots that gives a total vision of both the functionality and the user interface elements the users interact with.

The application is developed based on modern Google Cloud technologies that offer a number of advantages as a simple pricing model and the cheapest on the market, offers robust data privacy and security features, performance, whether in terms of latency, speed, processing power or redundancy in networks, according to the link found at index [5] *Google Cloud Advantages and Disadvantages* from *Bibliography*. The main technologies used belonging to the cloud domain are App Engine, Cloud Datastore, Cloud Storage Buckets and the source code of the realized elements is represented by Html and CSS programming languages for frontend side and Javascript, Ajax and Node Js programming languages for backend side.

My personal opinion related to the application created called the Laboratory Management Platform is very good being personally satisfied with the development process that produced this useful application because it meets the needs of students for an application that contains their entire student situation during a semester.

I designed the application to be easy to use and to have a friendly interface to be that friend of each student who presents and notifies him about exams grades and attendance points a few clicks away and is a way to communicate with teachers about possible misunderstandings about the grades obtained. I hope that one day the applications like the one I developed will be widely

used to give immediate access to data and I think it is a better way to do this process online than in the classic form in which a paper is used on which each student writes his name.

Bibliography

[1] Building Your Next Big Thing with Google Cloud Platform - S.P.T Krishnan and Jose L. Ulgia Gonzalez

Link: <https://www.pdfdrive.com/building-your-next-big-thing-with-google-cloud-platform-d24624408.html>

[2] Cookies Overview

Link: https://www.w3schools.com/js/js_cookies.asp

[3] Google App Engine

Link: <https://cloud.google.com/appengine>

[4] Google App Engine Overview

Link:

<https://cloud.google.com/appengine/docs/flexible/nodejs/an-overview-of-app-engine>

[5] Google Cloud Advantages and Disadvantages

Link: <https://synaxa.io/2019/06/17/pros-cons-google-cloud-platform/>

[6] Google Cloud Authentication

Link: <https://cloud.google.com/docs/authentication>

[7] Google Cloud Platform in Action - JJ Geewax

Link: <https://www.pdfdrive.com/google-cloud-platform-in-action-d186006829.html>

[8] Google Datastore Overview

Link: <https://cloud.google.com/datastore/docs/concepts/overview>

[9] Input Autocomplete Functionality

Link: https://www.w3schools.com/howto/tryit.asp?filename=tryhow_js_autocomplete

[10] MVC Architecture Overview

Link: https://www.tutorialspoint.com/mvc_framework/

[11] Node Js Nodemailer Package

Link: <https://nodemailer.com/about/>

[12] Node Js Promises

Link: <https://www.guru99.com/node-js-promise-generator-event.html>

[13] Side Navigation Menu

Link: https://www.w3schools.com/howto/tryit.asp?filename=tryhow_js_sidenav