Inspiring Excellence

CSE470 : Software Engineering
Project Report: SRS

Project Title :

ScholarSync: A Post-Grad Grooming Tool

Group No: **10**
Section : **08**
Summer 2023

Tasnia Juha (20101486)
Md Fardin Rahman Ami (20101549)
Sultana Marium Trina (20101059)

Table of Contents

Introduction

Our website or web application, *Scholarsync*, is designed to assist post-graduate students in their future pursuits. In the modern world, knowledge is dispersed across different places. To access specific information, we are forced to visit multiple websites, a process which is inefficient and time-consuming. To simplify the process of decision-making and information-gathering, we have developed a platform called *Scholarsync;* a "Post-Grad Grooming Tool." that integrates diverse sets of data and interactions related to different fields. By creating connections among these diverse elements, it facilitates the sharing of mutual interests and insights.

1.1 Purpose

The purpose of our website is to serve as a comprehensive platform catering to both students and professors, offering a range of valuable services. For students, it streamlines the process of discovering suitable job opportunities, scholarships, and courses to enhance their skills. By actively engaging in discussions and queries, they can establish connections and expand their network. The platform accommodates two user categories: basic and premium students. Premium users enjoy enhanced access to detailed information and features upon purchasing a premium subscription. Professors, on the other hand, can leverage the platform to share posts about research and teaching opportunities, benefiting from its information-sharing capabilities. Ultimately, this platform has the potential to be a crucial resource for recent graduates, furnishing them with pertinent information, skill-building prospects, networking avenues, and pathways to success as they embark on their early careers.

1.2 Overview

Our project report is divided into **7** parts with specific subsections, each serving different purposes. The initial idea and aim is proposed followed by the process of development through meetings and sprint updates. Moving ahead, the viewpoints of the website are explored from different perspectives with images for better understanding. Next, certain questionnaires are presented and solved in order to get a clearer retrospect of the scenario. By delving deeper into the walkthrough of the website step by step, a wider picture of it can be grasped. To make things more conventional, a use case diagram is presented followed by a relevant scenario.

Inception

The web application was built primarily based on certain stages, namely sprints, where each update and changes were added. The workload was divided into 4 consecutive sprints with an assigned scrum master for each sprint.

2.1 Planning Meetings

Scrum meetings form a vital component of the Agile Methodology. These gatherings consist of the **Daily Scrum meeting**, **Sprint Planning** and **Sprint Review**. Our team took turns on leading each sprint with deciding on particular features and updates. We conducted our regular Scrum meetings where each member discussed problems, updates and completed features to keep the rest of the team members updated. In the phase of Sprint planning, the respective scrum master assigned jobs to everyone in the team. Work was divided and an estimated deadline was set. In the sprint review, we gave each other informal updates via regular talks and concluded the end of a sprint. These meetings promoted teamwork, openness, and flexibility, facilitating incremental development and productive interaction among us. These meetings in general had a pivotal role in ensuring effective project implementation and the delivery of expected outcomes.

2.2 Sprints

Sprint 1 was mainly focused on building the blueprint of the website. We decided on choosing the Flask Framework with mysql database to store the required tables and necessary information. The main changes made in Sprint 1 is given below:

1. Connection with database
2. Value entry in specific table
3. Login and customer interface and Logout
4. Handling duplicate values for signup/login
5. Returning error message if password/email is wrong
6. Embedding urls corresponding to the courses
7. Search feature added
8. Sorting feature added based on alphabetical order.

Sprint 2 dealt with further update on features listed below:

1. The student login portal has been updated so that it checks the authentication of the correct user by displaying the logged-in user name on the home page.

2. Profile viewing has been introduced, allowing users to review their provided information on the Profile webpage. The webpage dynamically displays the logged-in user's given information based on the user.

3. A platform named Discussion has been established to allow users to share their ideas and ask questions. We created a dynamic and engaging conversation platform where students and lecturers could ask and answer questions.

4. A login module for students and professors has been created. It automatically leads to their corresponding sign-in pages based on the user category.

5. We added a cart for existing premium users of our service, where they can select any course and put it in their bucket to checkout later. (This is still in process)

In **sprint 3**, we tinkered the website deeper into achieving some more advanced features. These are mentioned below:

1. Now users are distinguished into two categories. Basic and premium users.

2. To become a premium user, a user needs to go through a payment gateway. If the payment is successful then the user will get premium status.

3. Premium users can access extra info on jobs, scholarships and faculties information.

4. A user can cancel his subscription of being a premium user by canceling the payment access.

5. In the discussion platform, users got distinguished. Now specific user name and post time is mentioned dynamically.

6. In Professor UI, a professor can post any research plan/ provide any opportunity of doing PhD and research work.

7. The students can see the posts and can contact the professor if they feel interested.

For **sprint 4**, the idea was to wrap things up by adding the final touch which was configuring and introducing the admin panel and some additional features. The details of the sprint are as follows:

1. The Admin Panel is created
2. Admin can login
3. Admin can add/remove jobs
4. Admin can add/remove scholarships
5. Admin can add/remove users
6. Admin can add/remove professors
7. Admin can logout
8. A feature of creating a chat room was added
9. Students can now interact with professors live by entering the specific room code

Viewpoints

In our project, we dealt with the non-participant side, student side, professor side as well as the admin side. Each side holds a completely different perspective with unique features to stroke upon.

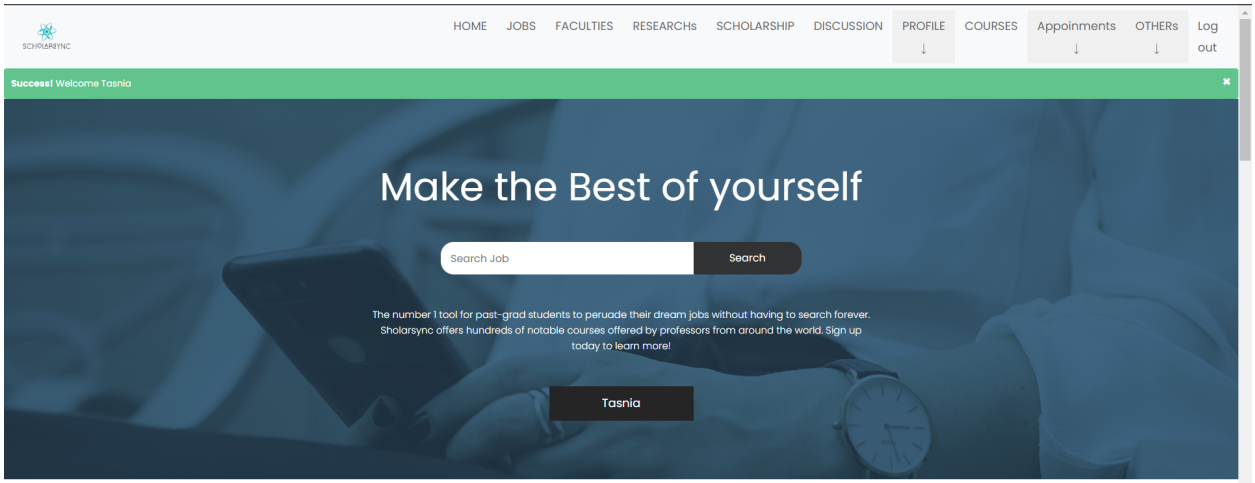
3.1 Recognizing Multiple viewpoints

Non-participant

The non-participant is someone who did not create any account but just came to visit the website just as a visitor.

Student

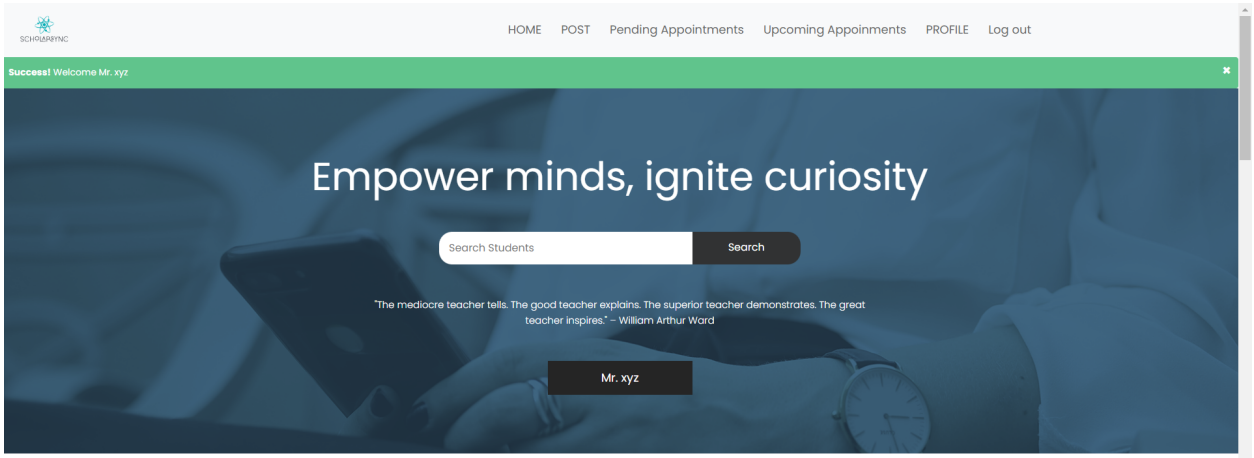
The student side view holds a completely different interface with newly added features in the navbar such as Research, Profile, Appointments etc. The main idea is for the student to interact with other students as well as with Professors and book relevant job opportunities. The student side has 2 versions: the **default** or **basic** version and the **premium** version which unlocks many extra features and opportunities



Marketing jobs

Professor

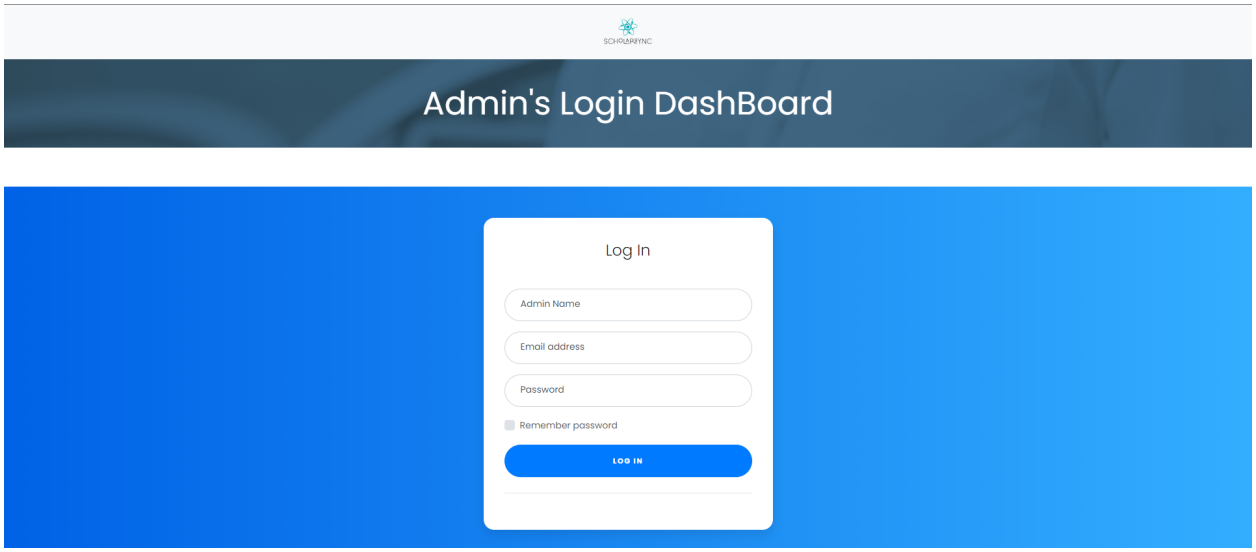
The professor side holds an interface with the necessary features required to put themselves out to the audience effectively such as Pending Appointments, Discussions etc.



Marketing Courses

Admin

The admin interface holds the key to managing the entire website and ensures that it runs smoothly.



Questionnaire of the Project

The project aimed to target a specific audience, delivering just the right amount of information when and where necessary. We tried to solve some possible problems based on the questionnaire below:

4.1 Questionnaires

- Why is it needed?
- What is the purpose and objective of the project?
- What are the benefits?
- Who are the users?
- Will it be cost-effective?
- Is any Interaction module (like discussion panel) available?
- How is the security ensured?
- How will it be useful to teachers and students?

4.2 Proposed Solutions

- To help post graduate students find job opportunities and/or relevant courses to develop their skills for future opportunities.
- To make the communication between students and professors smoother and bring forth jobs to job seekers right under their fingertips.

- Decreased unemployment rate and an increase in economic growth with higher educational qualification rates.
- Post graduate Students and Skilled Professors from different sectors.
- This is undoubtedly a cost effective approach since people would not have to travel distances to get a job or get tutored.
- Yes, A discussion section allows students to interact with each other.
- If one user signs into their profile, others cannot steal their information. Also, if there is any sort of bullying found in the discussion panel, the user can easily be dismissed by the admin
- Students will get to learn a new skill sitting at their homes and Professors would get a greater audience and thus higher recognition.

Non Functional Requirements

Non-functional requirements define the attributes that characterize the behavior, performance, and constraints of a software system rather than its specific functionalities. Unlike functional requirements, non-functional requirements describe how the system should perform and what qualities it should possess.

5.1 Listing Down non-functional requirements

Non-functional requirements play a vital role in shaping a software system that is not only functionally complete but also performs well, is secure, and meets the expectations of users and stakeholders. Some non-functional requirements of the web application are listed below:

1. **Performance**

   The response time or loading time was pretty fast, which is a good sign and helps avoid delays. Moreover, Gunicorn's multi-worker model and asynchronous capabilities enhance the performance of our application, allowing it to handle a large number of WebSocket connections efficiently.

2. **Security**

   Due to password hashing, even the administrator was unable to retrieve user password, protecting and maintaining the safety of the user.

3. **Reliability**

   The website was consistent throughout the whole testing process, enhancing efficiency by utilizing the available resources properly.

4. **Maintainability**

The administrator could easily update the information of the website without having to gain complicated knowledge regarding complex coding and just by pressing buttons.

5. **Usability**

   The application provides a simple user interface with predictable keys and buttons here and there for easy access. The user would not need to have sophisticated knowledge related to websites. Only general knowledge would suffice.

6. **Real-Time Communication**

   Socket.IO enables real-time communication between the professor and the students in the chatroom, meeting the requirement for live chat functionality.

7. **Scalability**

   The use of Gunicorn as the application server allows our project to handle multiple concurrent connections efficiently, ensuring scalability as the number of users increases.

Walkthrough

Here is a short manual of the whole website and its ins and outs to get a brief idea of the UI.

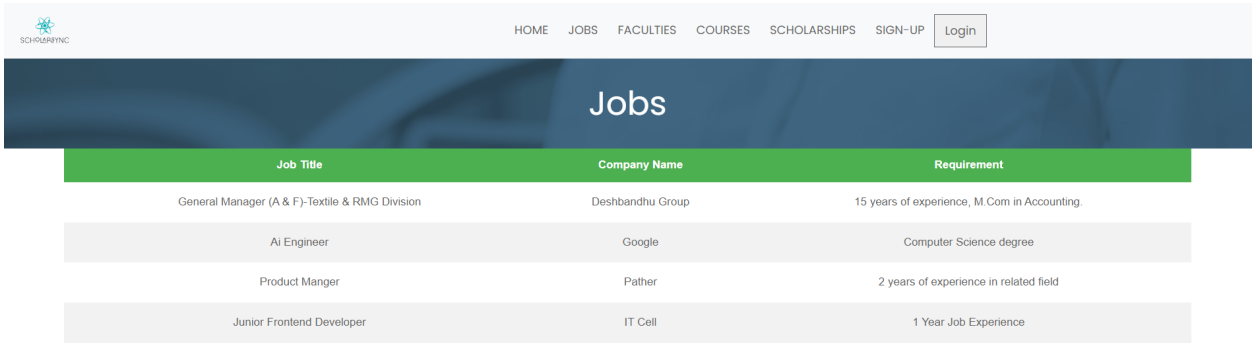
6.1 Website Manual

6.1.1. Homepage:

Both the student and teacher can browse to the homepage and can access the given tabs such as jobs, faculties, courses, scholarships according to their needs.

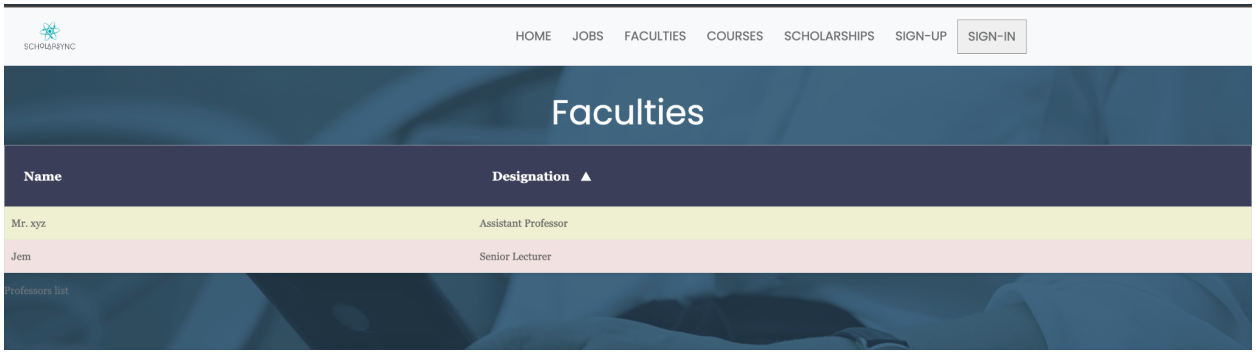
The functions of buttons present in the navbar are as follows:

(a) Jobs: To seek recent job opportunities, any random user can browse it. It also shows related information, and job title, deadlines, requirements etc.

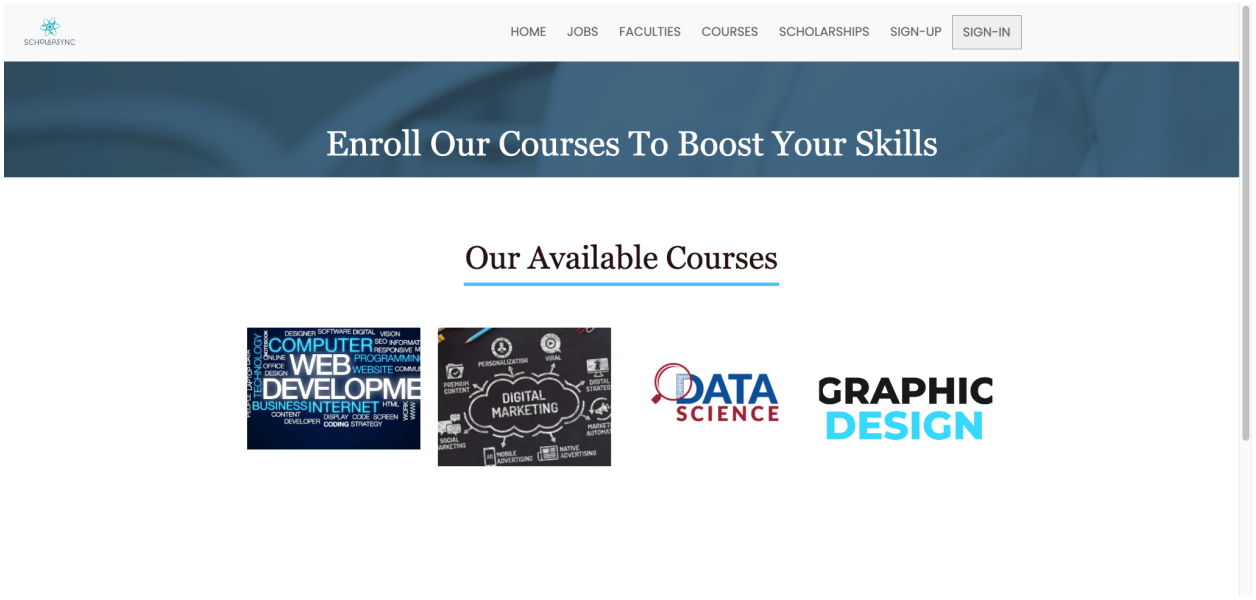


Job Title	Company Name	Requirement
General Manager (A & F)-Textile & RMG Division	Deshbandhu Group	15 years of experience, M.Com in Accounting.
Ai Engineer	Google	Computer Science degree
Product Manger	Pather	2 years of experience in related field
Junior Frontend Developer	IT Cell	1 Year Job Experience

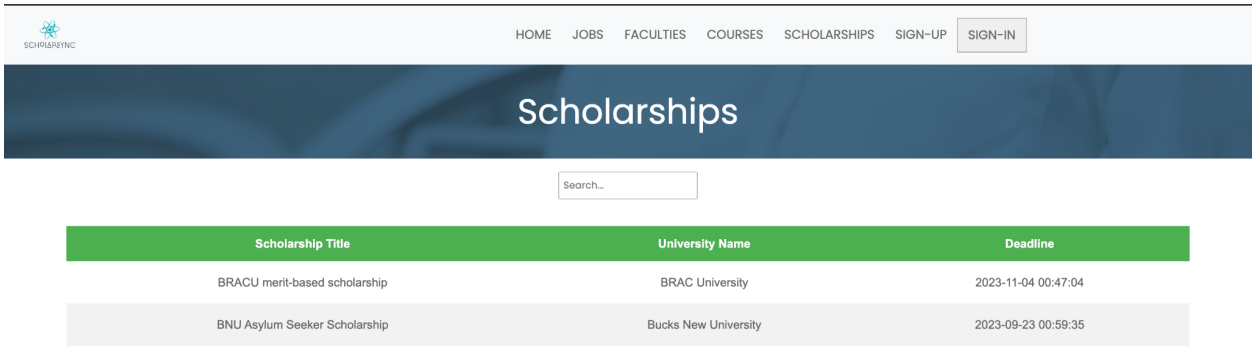
(b) Faculties: To get any information about the faculties or professors, this tab will redirect them there.



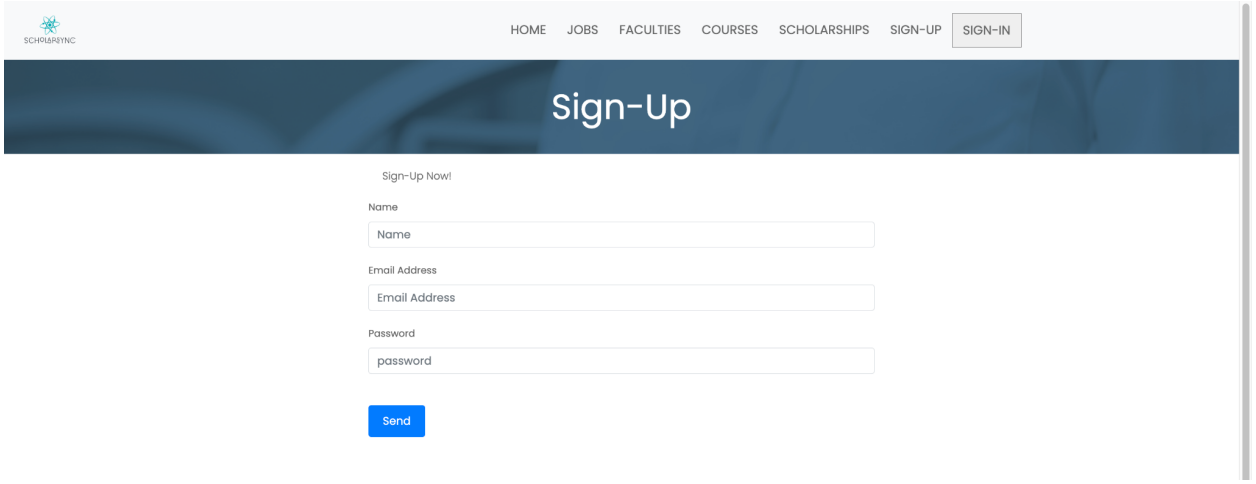
(c) Courses: This tab will display the courses that the website offers. Clicking on each would redirect the user to that particular url.



(d) Scholarships: This webpage will give the user information about the scholarship type and relevant data.



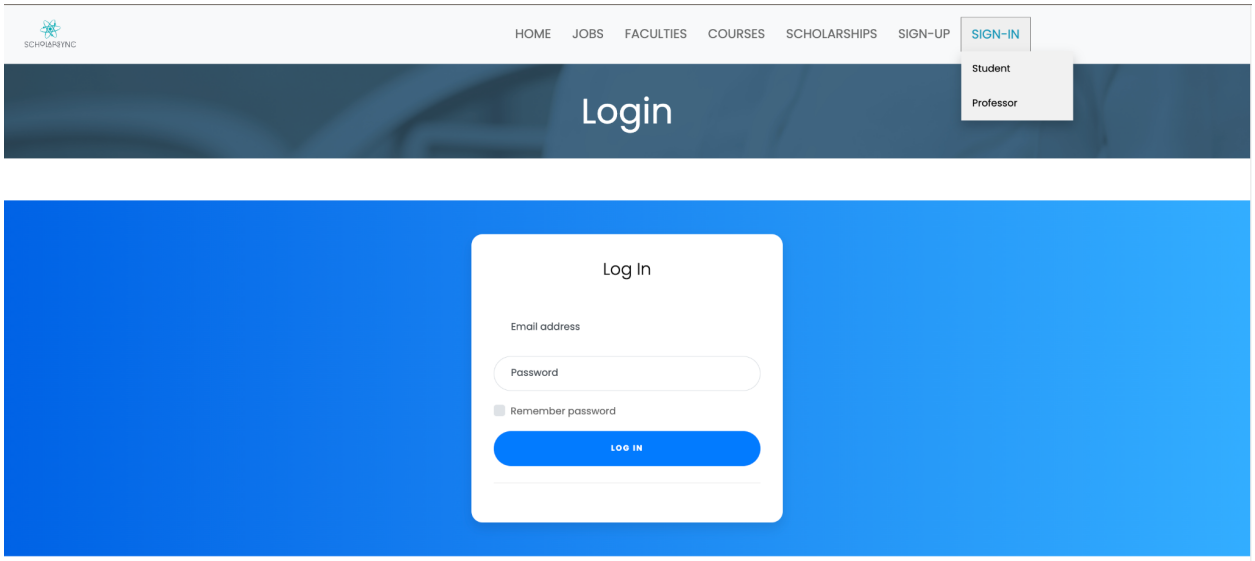
(e) Sign-Up: To access the other features and to use the utmost modules a student needs to register and create a profile. Through this registration, he will get authentication to access the website.



(f) Sign-in: To log in to the site and access one's own profile, one needs to login through his email and password. Sign-in has a drop down action that shows identical login options for students and faculties.

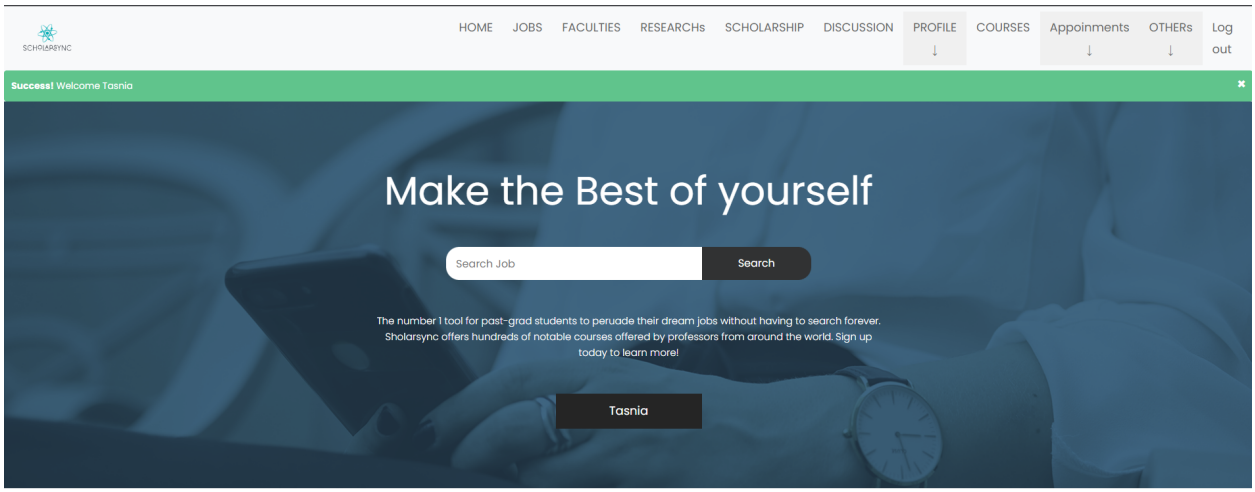
(i) User Authentication: To get logged in, it verifies the email address and password. It checks whether the user provided valid and registered data or not. If the data is valid, then the user can log into it. Otherwise, it prompts the user a invalid data.

(ii) Current user identity: After logging into the website it keeps track of the users who are currently logged in to the site. It also keeps individual track of tasks that are done by a user.

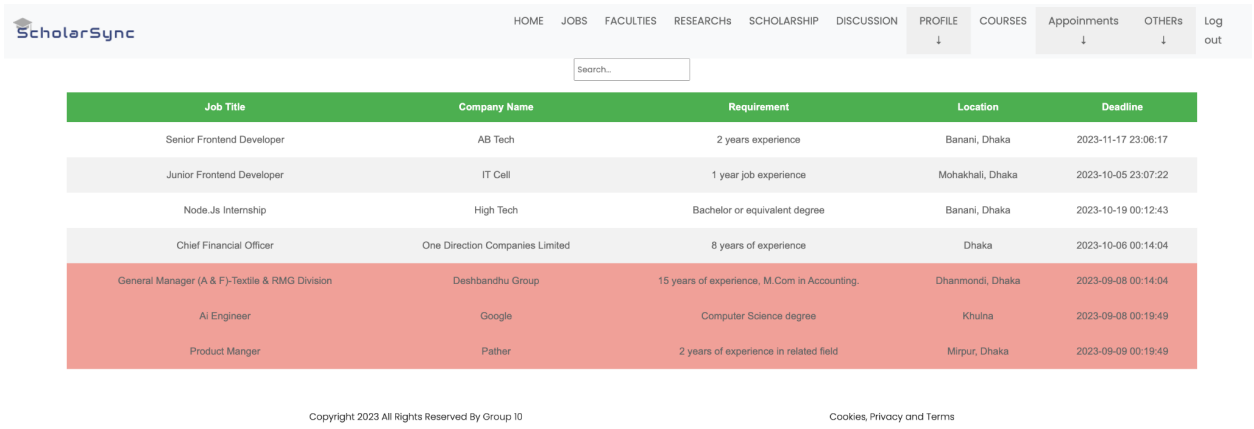


6.1.2. Student's Interface:

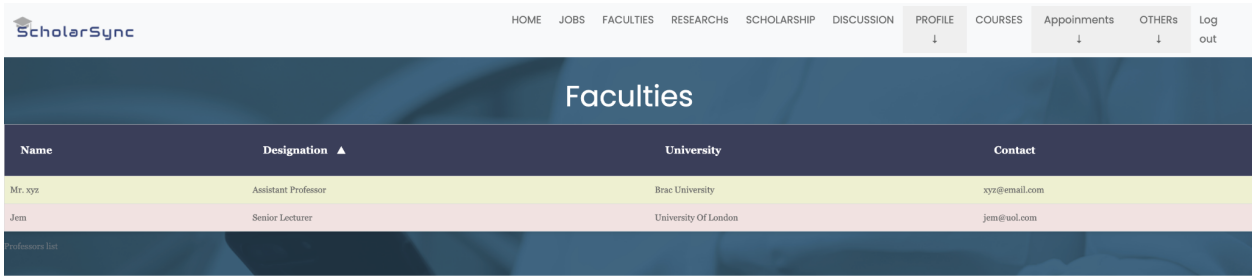
After logging in to the site as a student, a user can view this frontend site that gives a highlights by showing the user name and Gives multiple tabs to access distinct features.



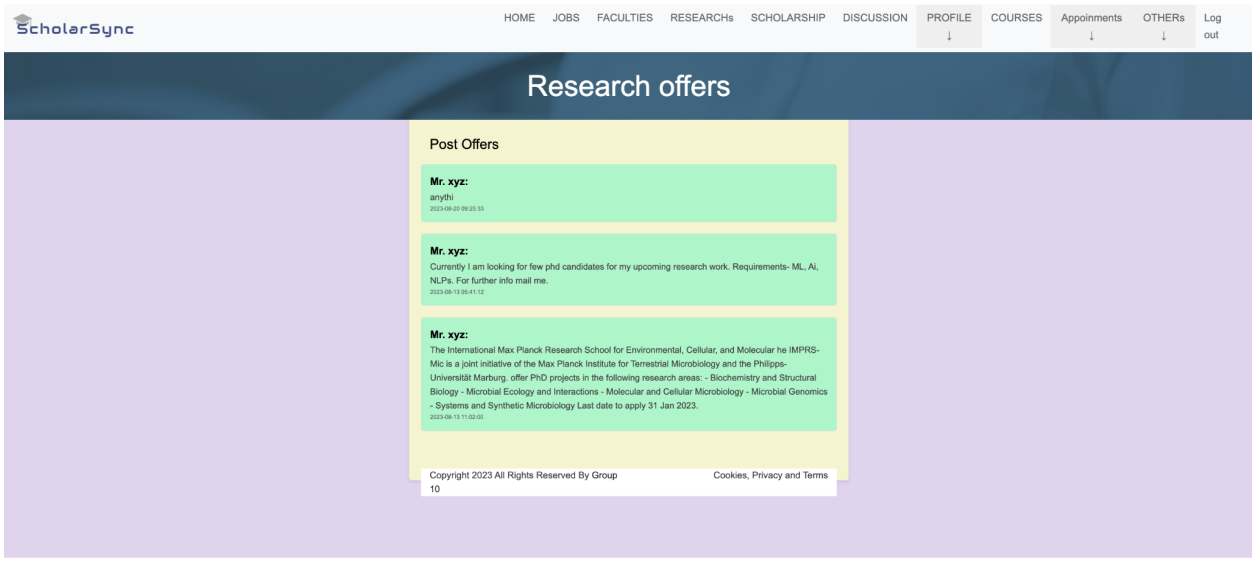
(a) Jobs: Here the student can seek jobs related information. Here the table is more functional. It can be sorted through any column head and also can be searched through any particular information. If the deadline is over, then it dynamically highlights the rows. Premium users can access the entire facilities.



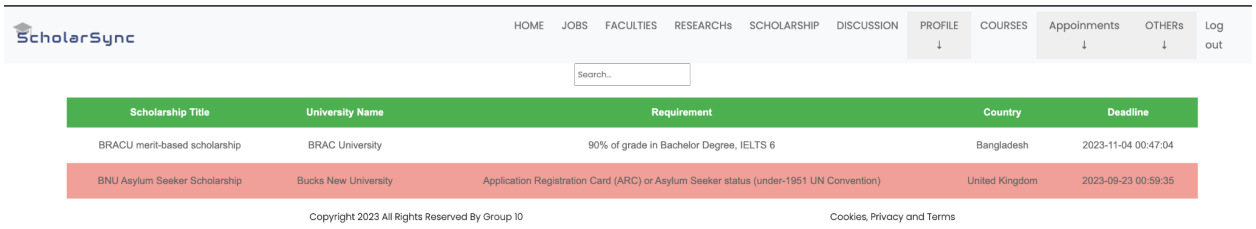
(b) Faculties: After getting logged-in and becoming a premium user, a student can browse the faculties section to get the available faculty's detailed information or contact ways.



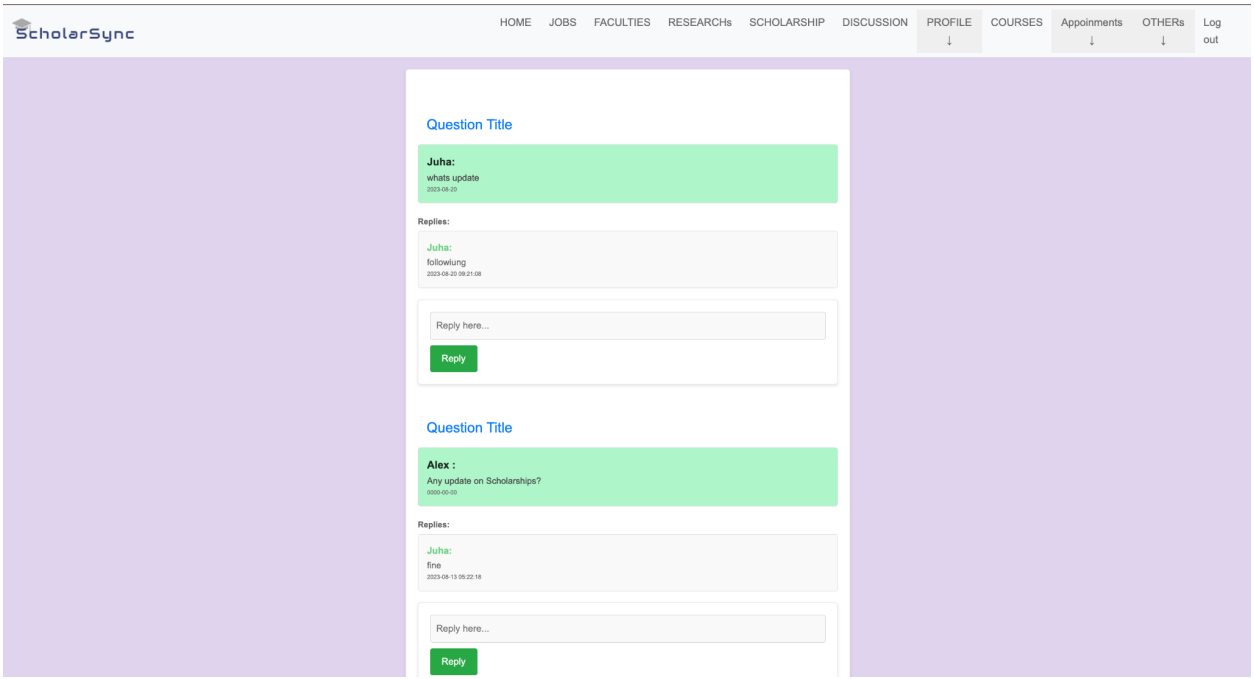
(c) Researches: If any professor or faculty member gives any update related to any project or research work, then it will be shown here. Interested students can follow up these updates.



(d) Scholarships: After signing in to the site, a student can now access more and other relevant information regarding the scholarship requirements, university name, deadline and so on.

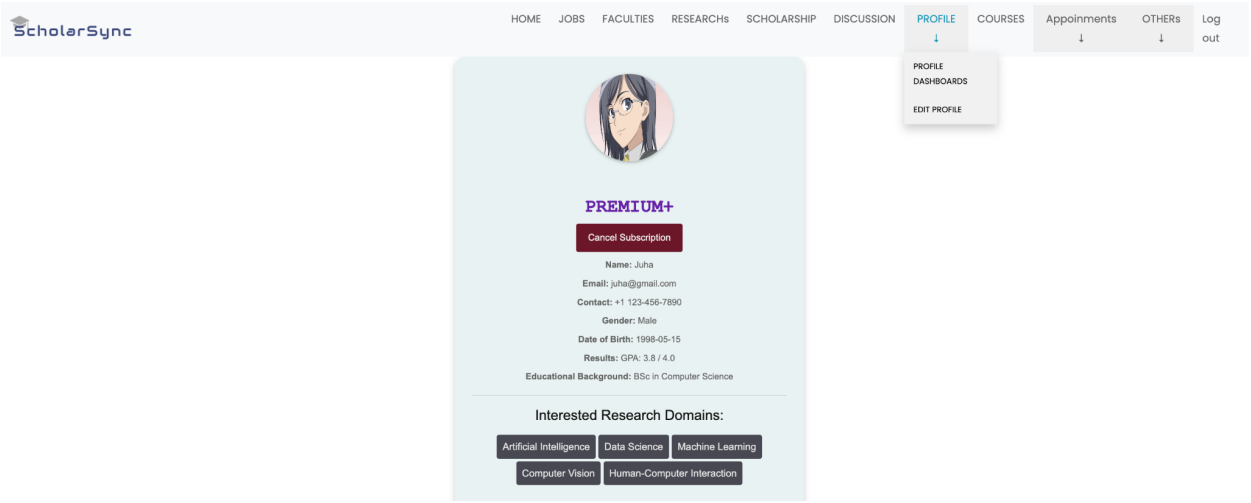


(e) Discussion: To interact with other students and to discuss any topic, students can post here. Other students can see the posts and reply to their queries. It's a multi-user interaction platform

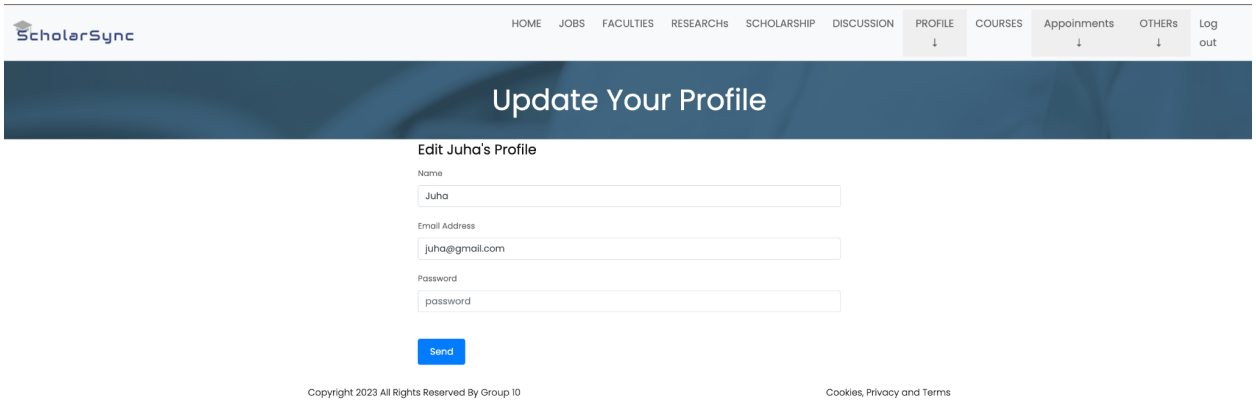


(f) Profile: To check a user's profile and to edit them a student can go through this profile tab.

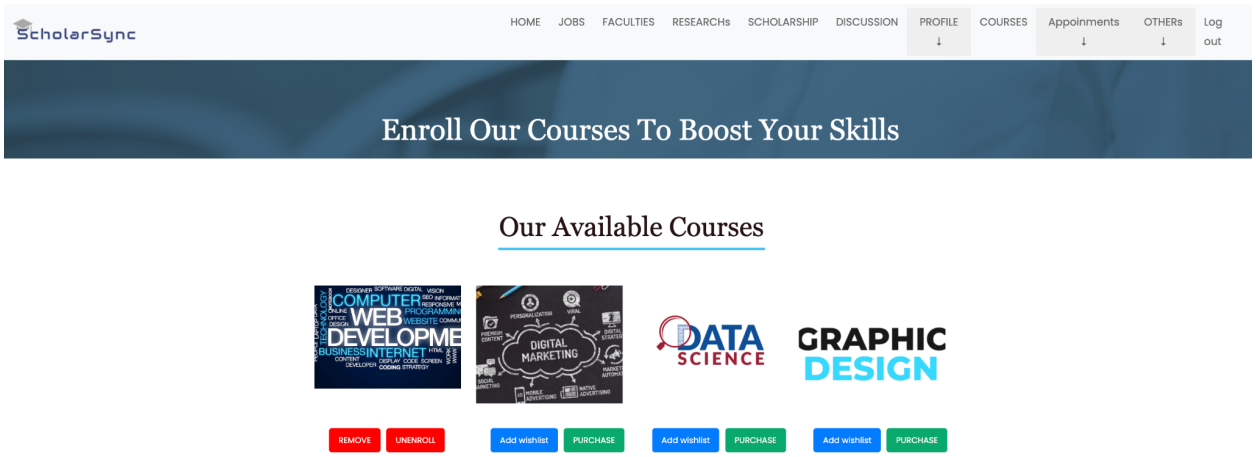
(i) Profile dashboard: It shows the user's subscription mode (basic or, premium), illustrates user's given information.



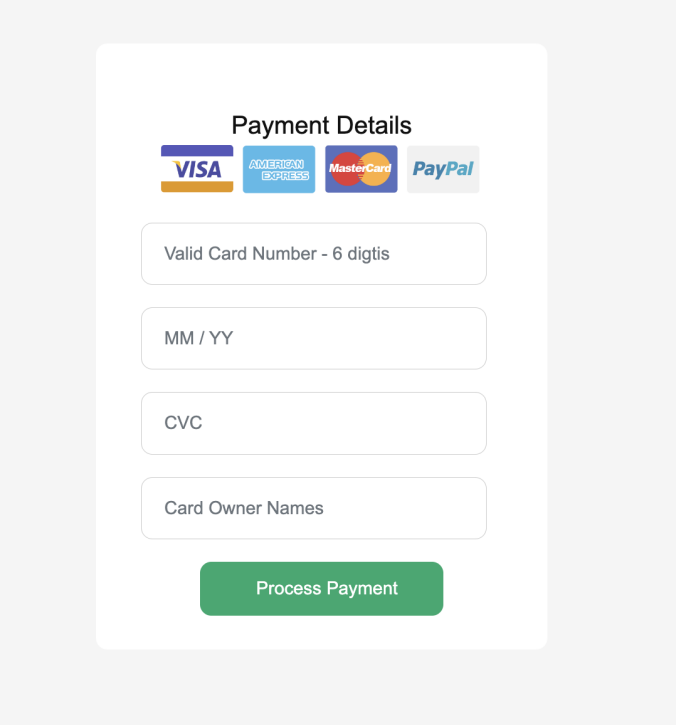
(ii) Edit profile: If a user wants to edit any information of himself, he can update the information by providing updated data here.



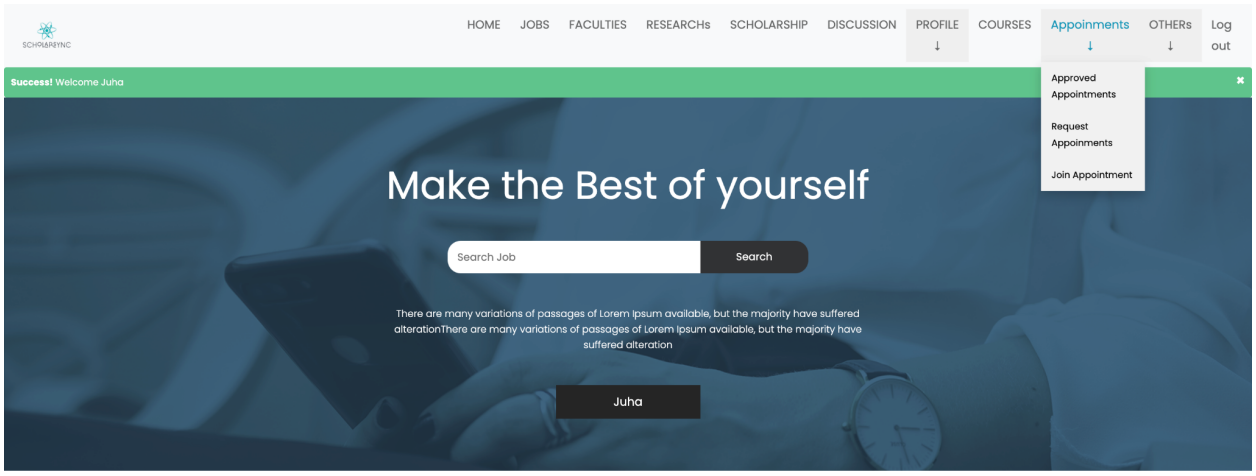
(g) Courses: From the course tab a student can purchase courses and also can add to the wishlist to keep them to a separate page.



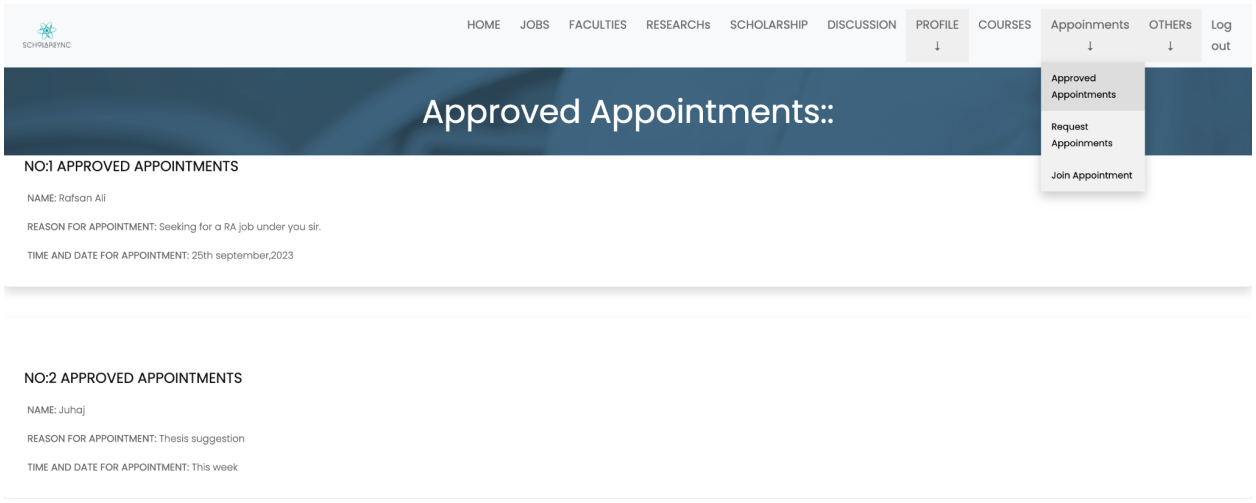
(i) Purchase course: To enroll in the course, a student needs to proceed through a valid purchase system. Otherwise the purchase request will be canceled.



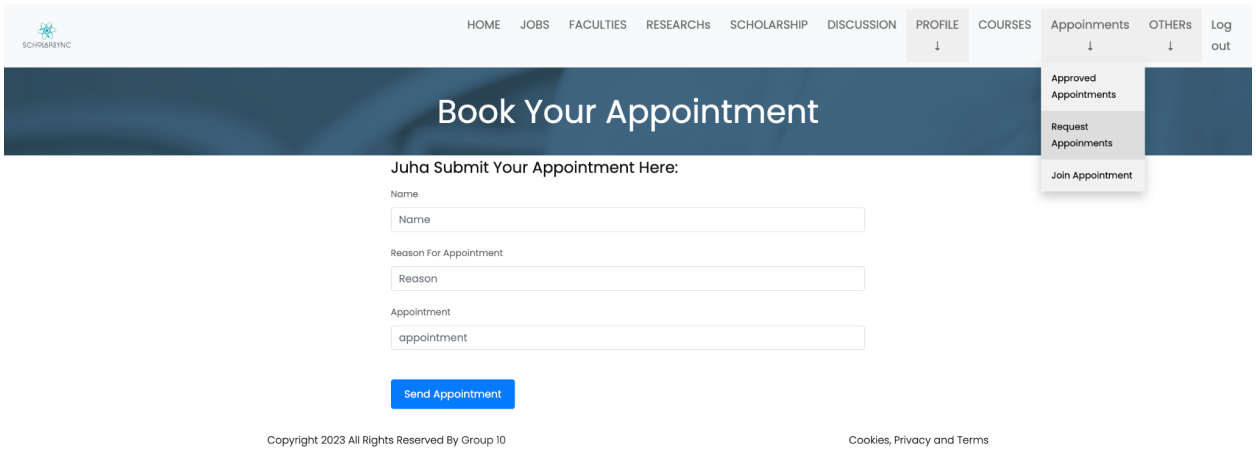
(h) Appointment: A student can use this feature in order to book for an appointment , check the update of the appointment from faculties and join appointments where they can talk in real time with faculties.



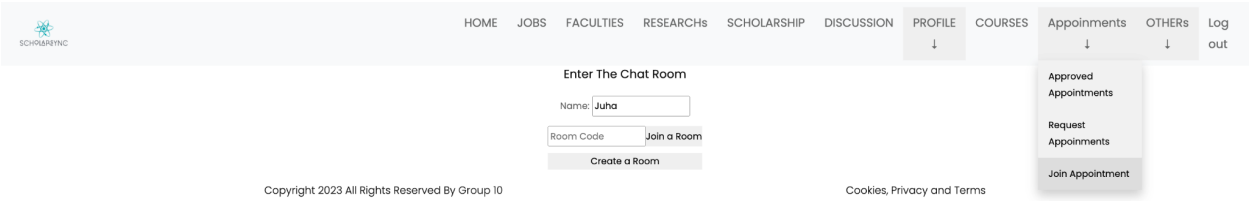
(i) Approved appointments: Here the student can see his approved appointment list that he requested for.



(ii) Request Appointment: If a student needs to consult with any professor then he can request for an appointment using a form.

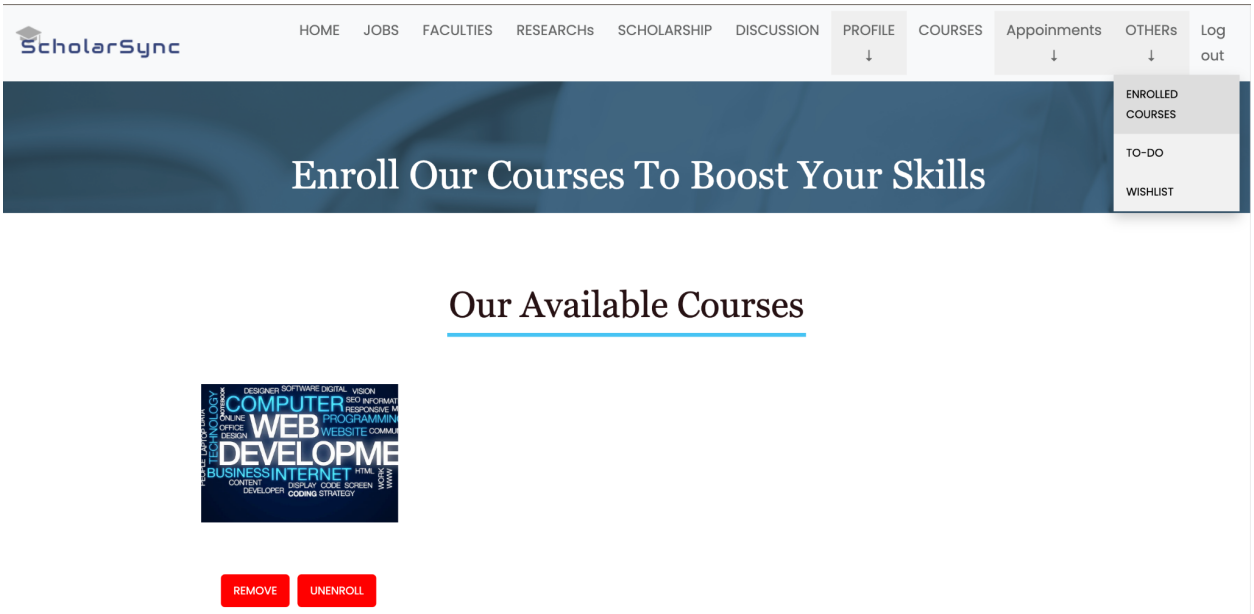


(iii) Join Appointment: After the professor approves the appointment request and gives a code to join a room, students can join an appointment to interact with the professor and talk in real time with the professor.

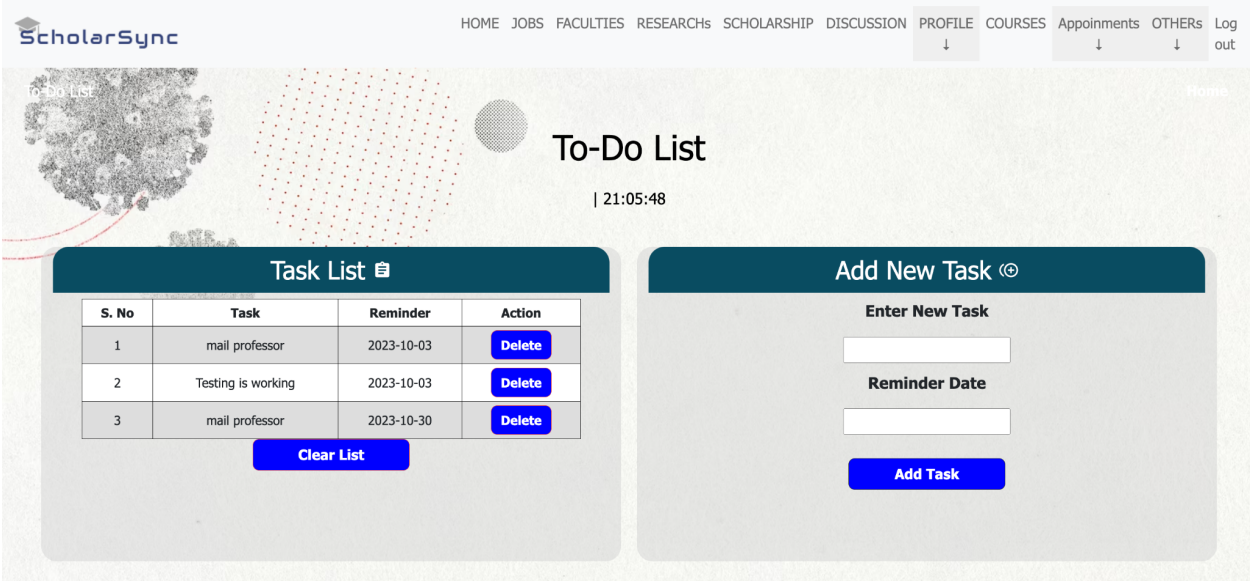


(i) Others: In this tab there are three sub-tabs that contain enrolled courses, to-do and wishlist.

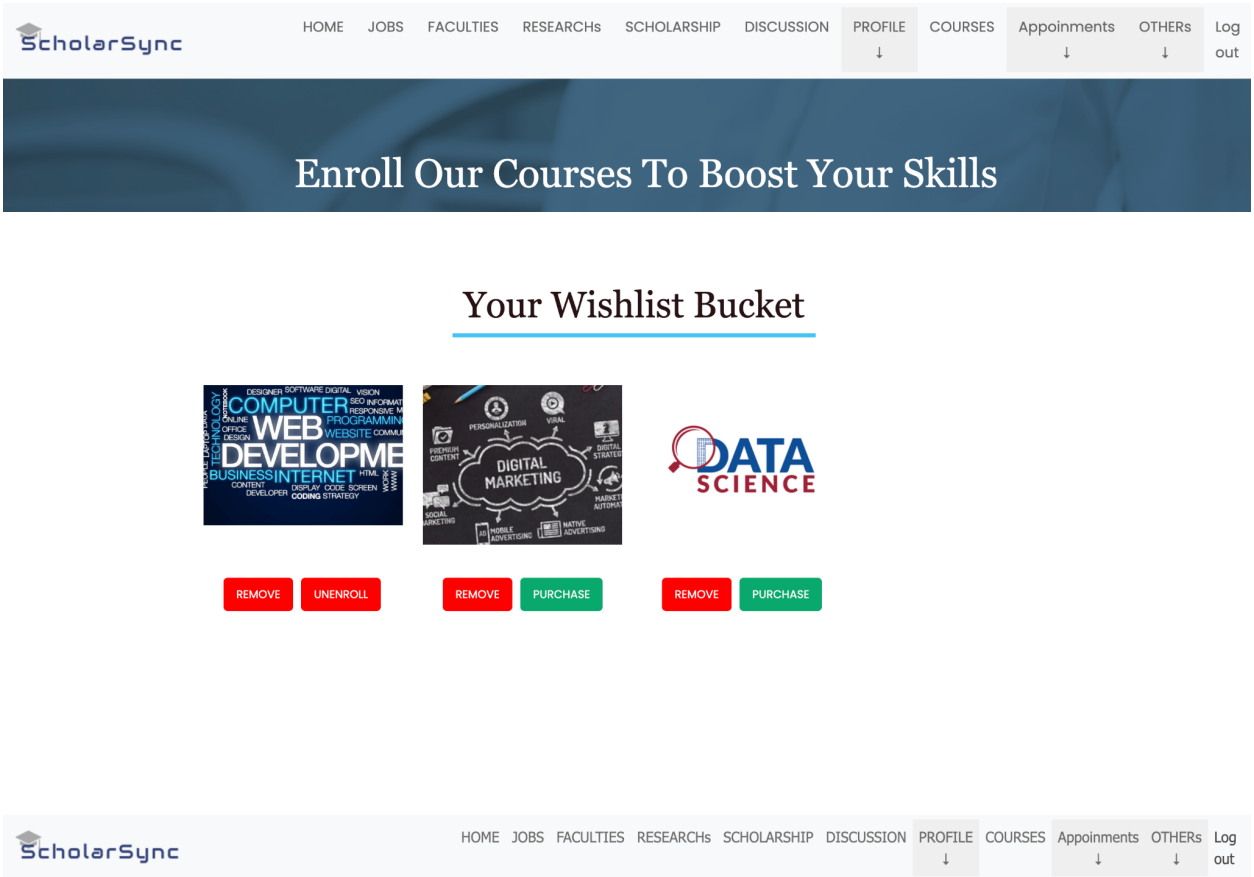
(i) Enrolled courses: All the purchased courses of the students will show here and they can access the courses.



(ii) To-Do: A student can add any pending task with a reminder date here. This to-do list can be updated and deleted according to the need.



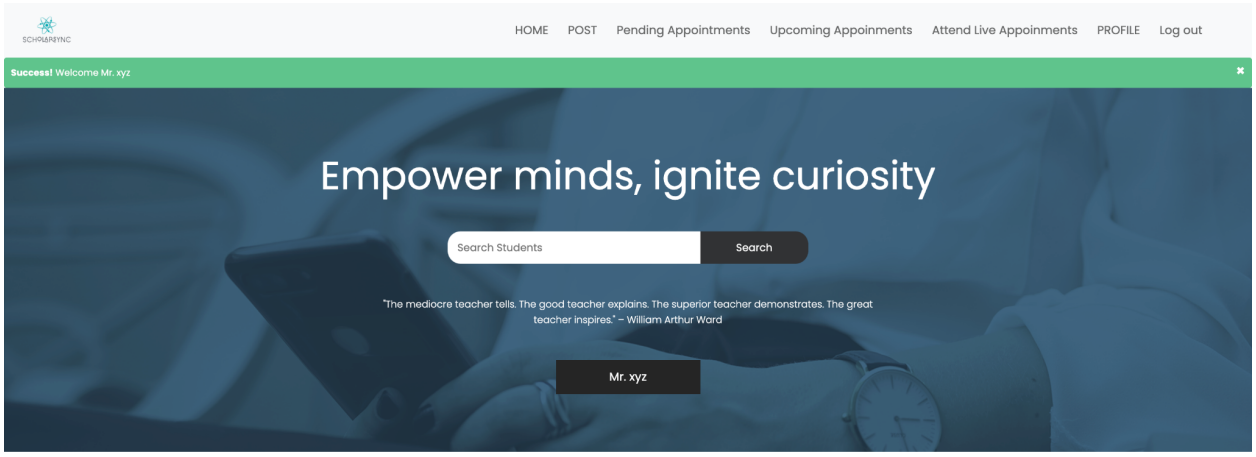
(iii) Wishlist: A student picked any course to keep them separated to a wishlist. Those courses list will show here separately.



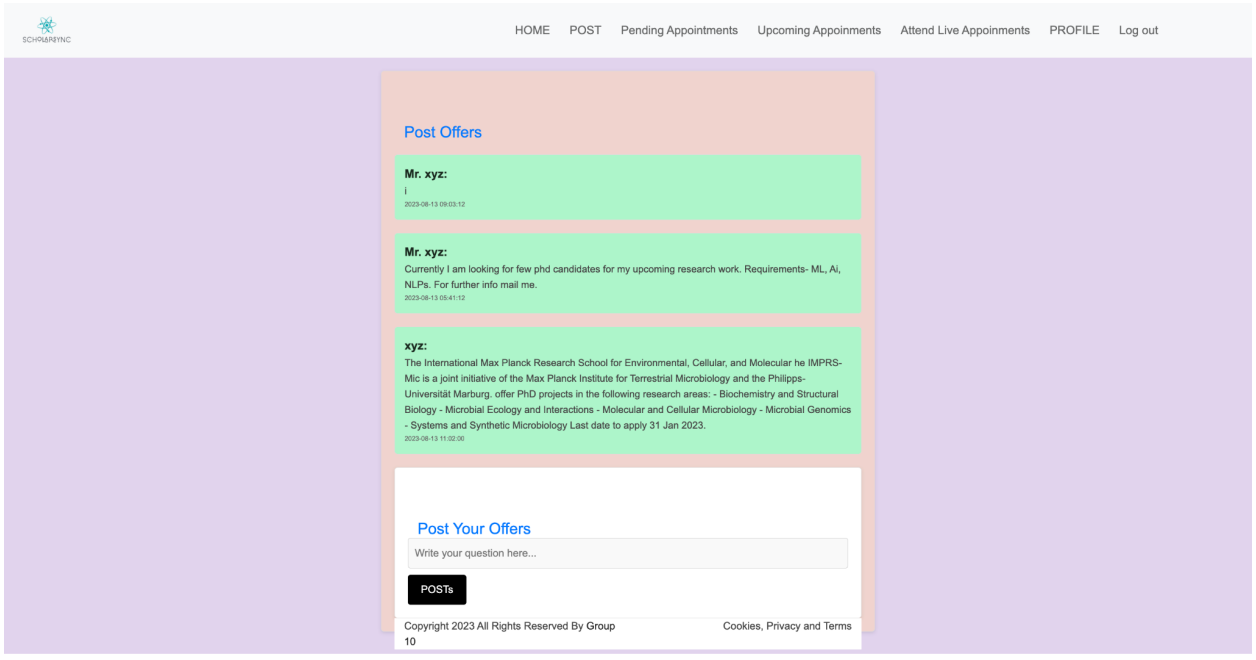
(j) Log out: After clicking on the log out, the user will be returned to the homepage of the website and can browse as a guest user.

6.1.3. Professor's Interface:

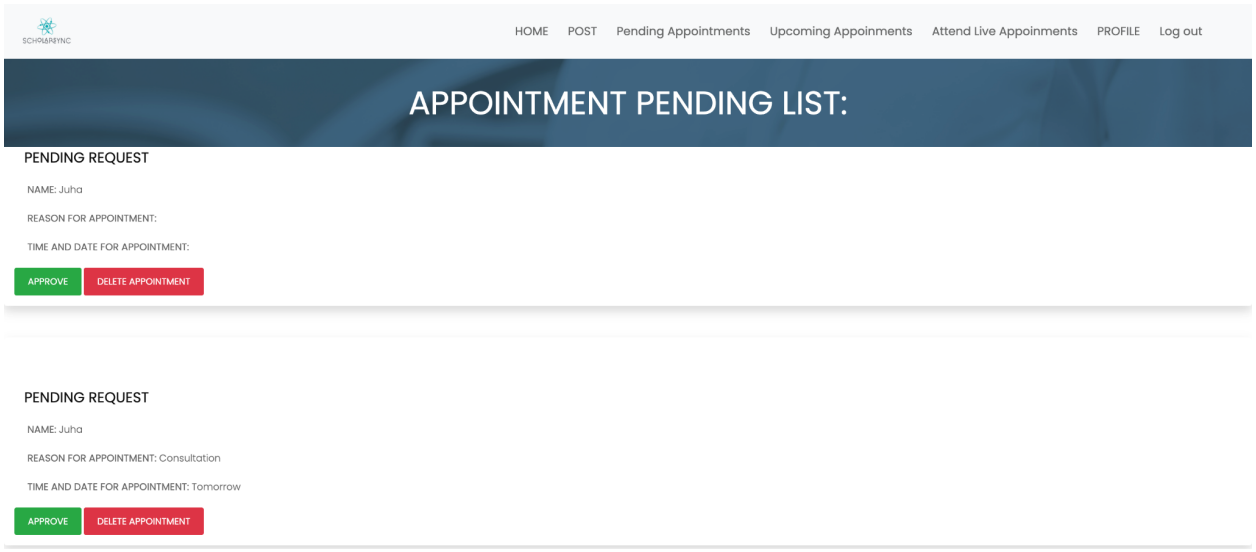
After a professor logs in to the platform, the webpage will give him access to posting, responding to appointments, upcoming appointment events, and his profile.



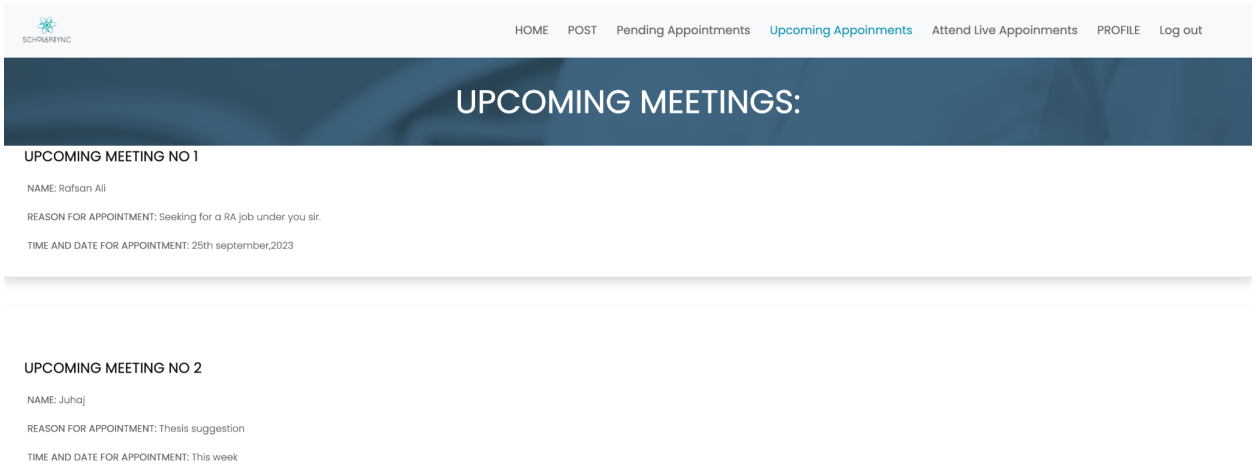
(a) Post: For any updates or, to offer any vacancies of RA/TA/PhD fields, a professor can post here with his or her own identity and datetime.



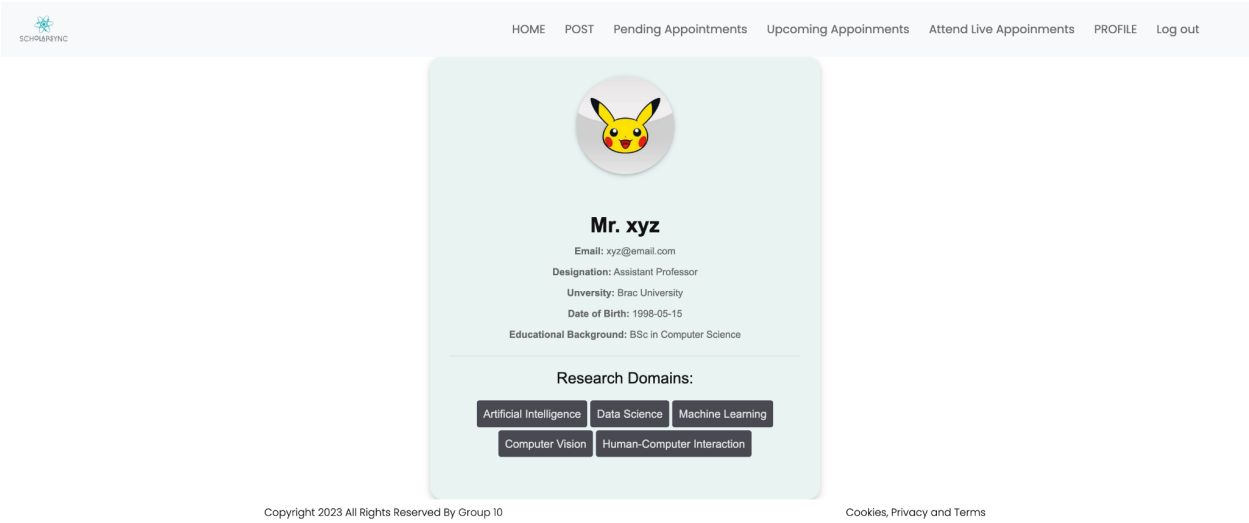
(b) Pending appointments: A professor can respond to the appointment requests of students. He can either accept it or deny it.



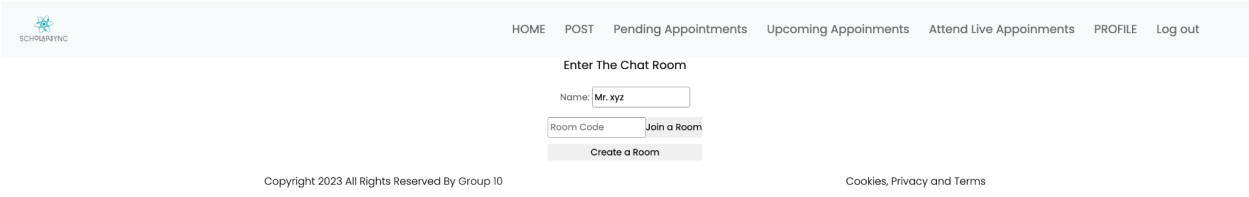
(c) Upcoming appointments: A professor can see all the accepted appointments and upcoming dates here.



(d) Profile: A professor can check his profile and give information here. It is a short section about a professor.

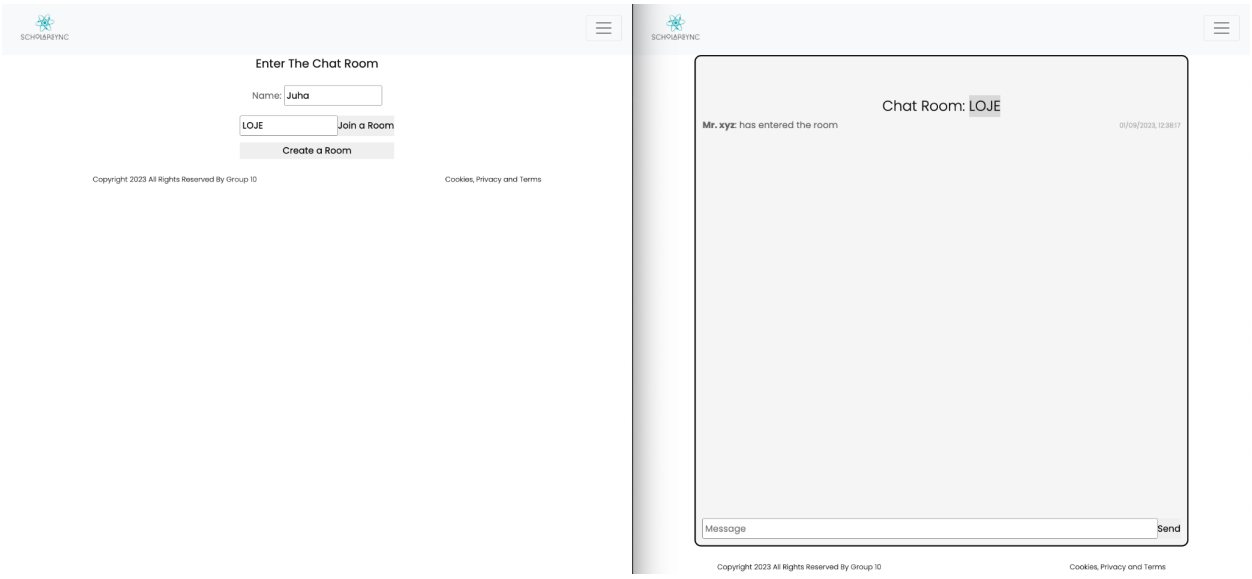


(e) Attend Live Appointments: A professor can attend live appointments by creating a chat room to interact with the students.

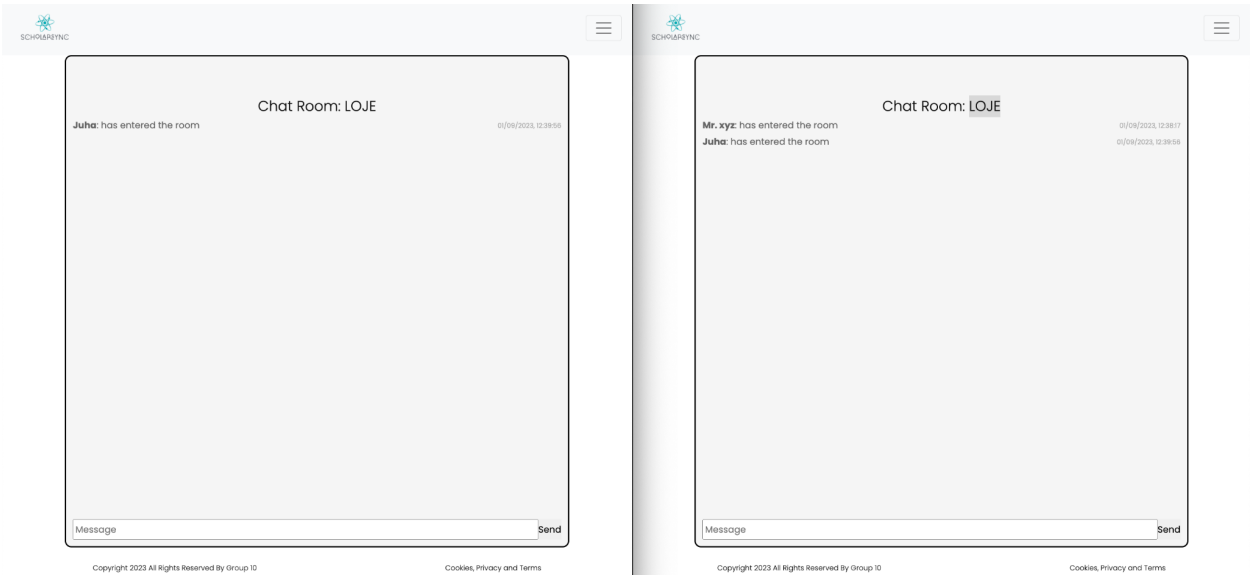


6.1.4. Chat Room Interaction from both Students' and Professors' Ends:

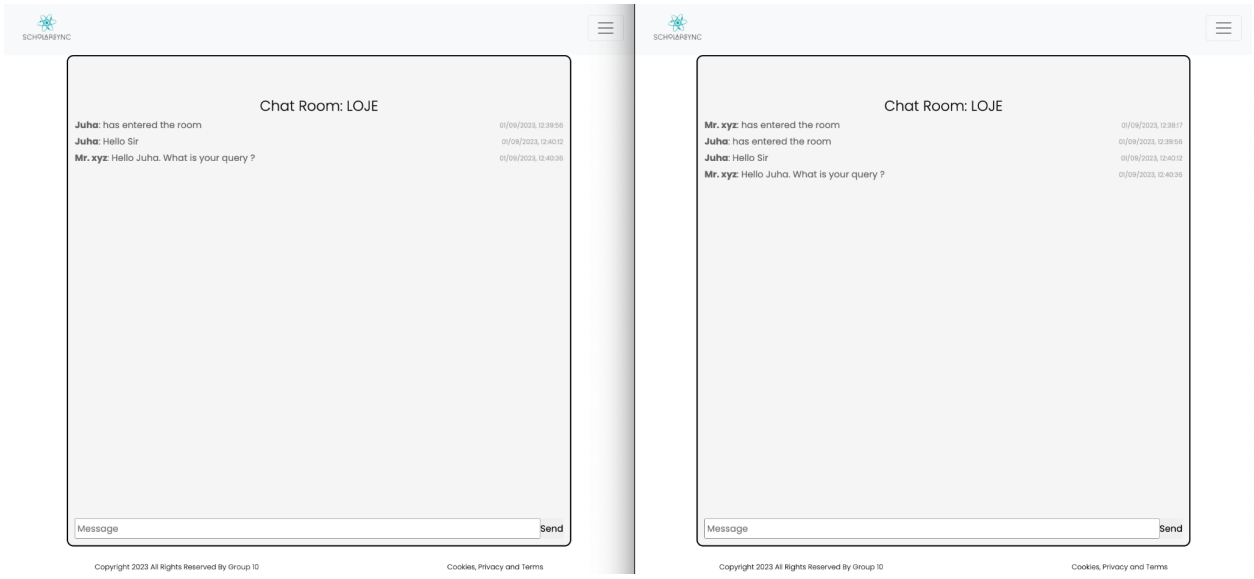
(a) Create and Join Chat Room: Once the professor loads the attend live consultation drop-down, it directs him to the user specific Chat Room Interface. From here, they can create a chat room. Once the chat room is created , it'll generate a unique CODE that the student will need to join that particular chat room.



(b) Interaction from both sides: Once the student joins a chat room with unique code, it'll be visible to the professor's side that -"the name"(student) has joined the room. Also, they both will be able to see the live timing of their interactions.

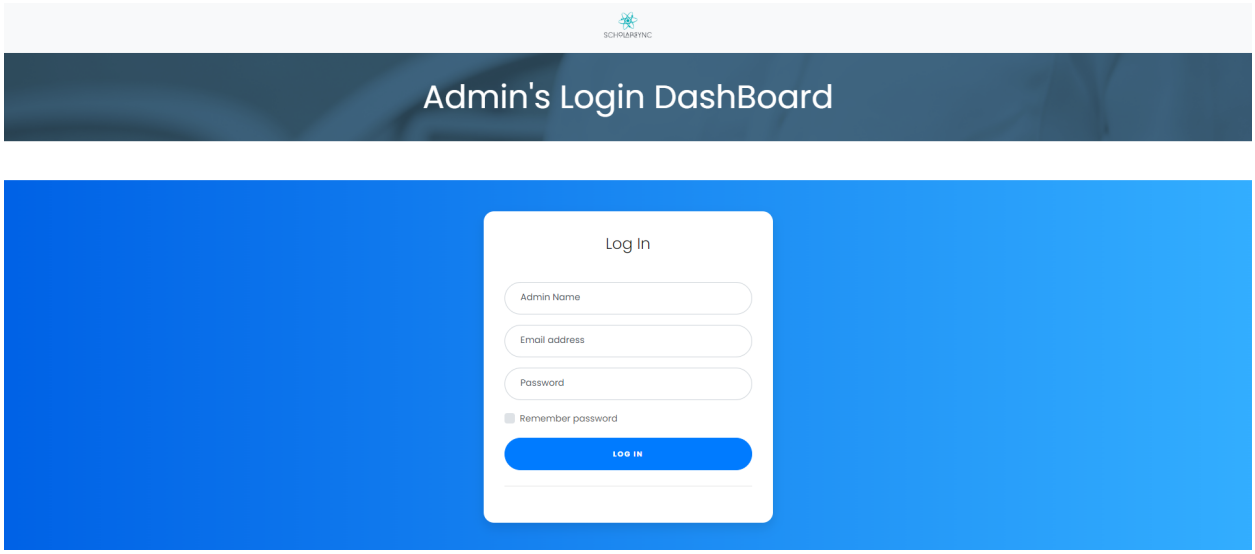


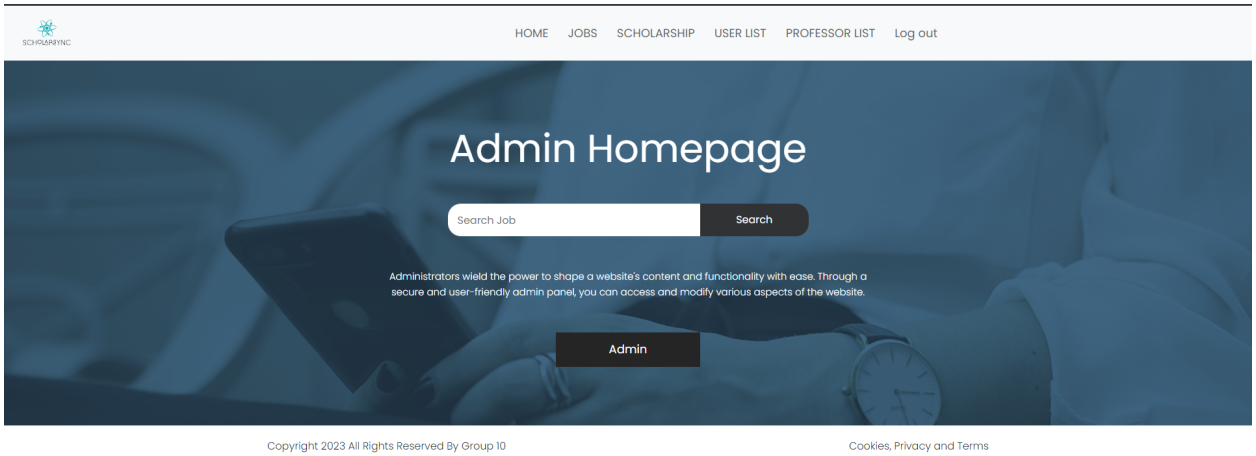
(c) Live Chatting: The messages sent from student or professor's side , will be visible to both of them so that they can share their thoughts and resolve queries.



6.1.5 Admin Panel:

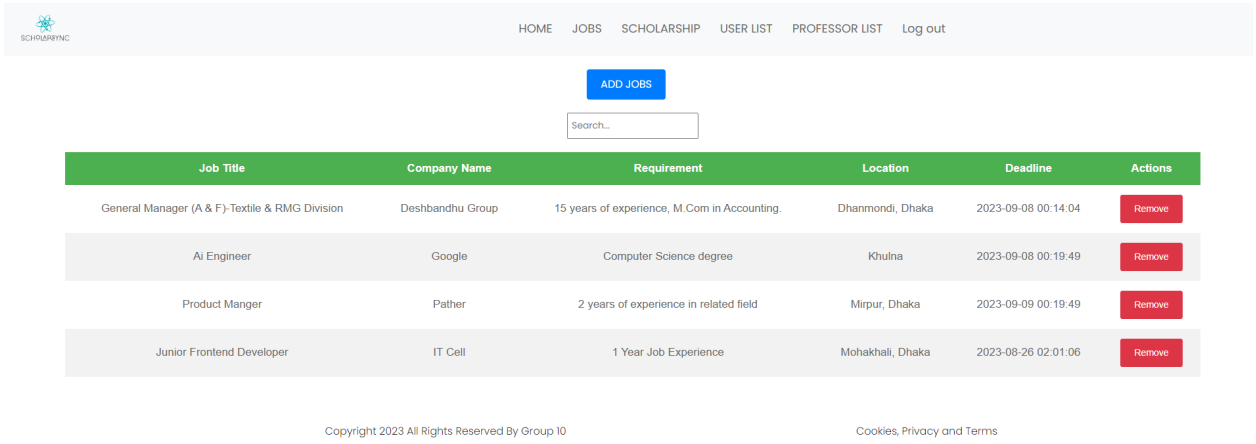
An admin can access and modify a few user data and check the overall registered users here. The link http://127.0.0.1:5000/logina takes a user to the admin login panel.



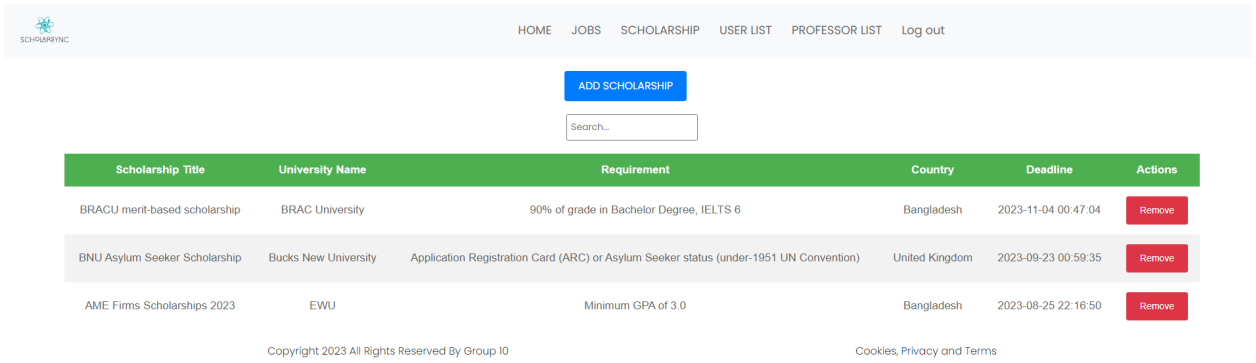


After logging in, he can access several navbar panels.

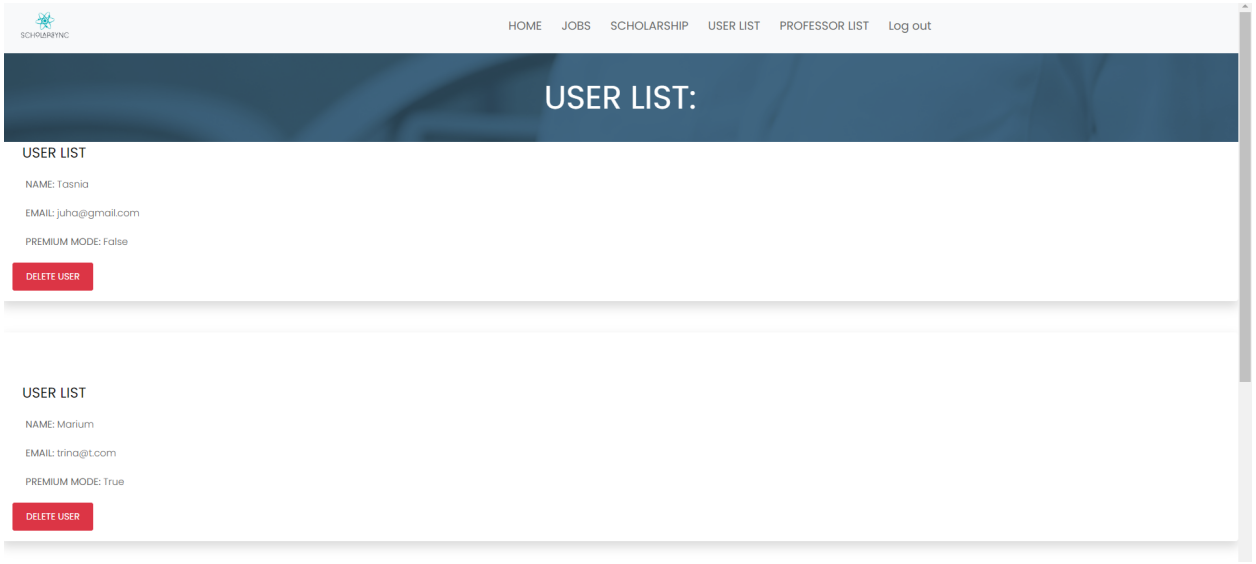
(a) Jobs: The current job lists will show here. By clicking on 'Add Jobs', an admin can add a particular job with its requirements or even 'remove' an entire Jobs column.



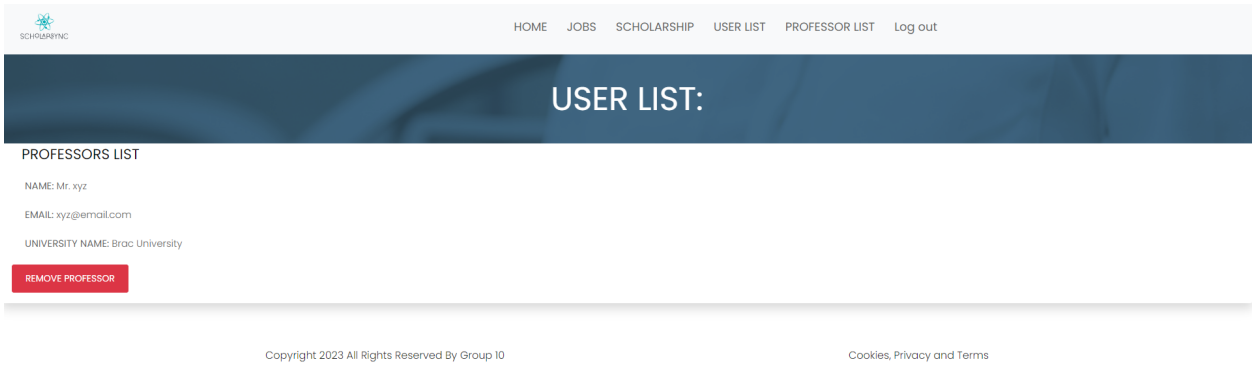
(b) Scholarships: Similarly the current available scholarships lists will show here. By clicking on 'Add Scholarship', the admin can add a specific scholarship update with its requirements or even 'remove' an entire column.



(c) User-list: The registered student users list and their information will show here. By clicking on ‘delete user’, an admin can ban a user from accessing the website and login features.



(d) Professor-list: The list of registered professor users and their corresponding information will be displayed in this section. By selecting the "remove professor" option, an administrator has the ability to prohibit a user from accessing the website and its associated login functionalities.



Data Model

Data modeling is an important task during the software development phase of information systems, commonly relying on database management systems for data storage. The outcome of this process is the data model. This configuration is frequently depicted visually using the UML or **Use Case Diagram**.

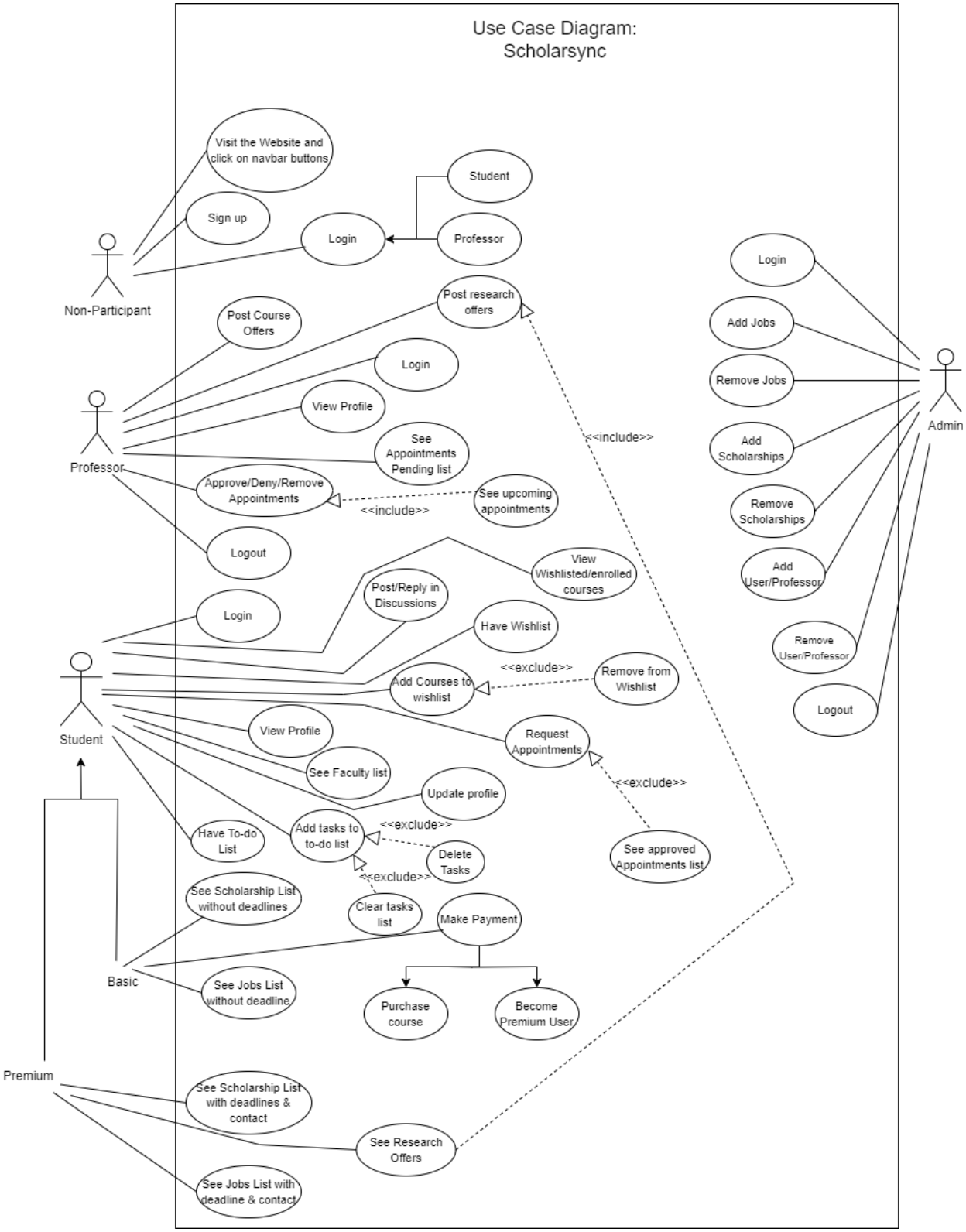
7.1 Use Case Diagram and Scenario

Since our website primarily consists of 4 actors, the proposed scenario should deal with the admin being the secondary actor. The primary actors are the non-participant, professor and student.

The non-participant is just a spectator who can only see the available options and press navbar buttons to get the idea of the website. If interested, they may then sign up and create an account to login and access the features available.

There is a generalization of the student actor which distributes to: *basic student* and *premium student.* The student actor can perform the following tasks as depicted in the diagram below.

Same goes for the professor. They can log in and schedule/approve or even deny appointments as per their convenience. The further details about the interaction is illustrated as follows:


Use Case Diagram:
Scholarsync
Visit the Website and click on navbar buttons
Sign up
Login
Student
Professor
Non-Participant
Post research offers
Post Course Offers
Login
View Profile
See Appointments Pending list
Approve/Deny/Remove Appointments
See upcoming appointments
<<include>>
Logout
Professor
<<include>>
Login
Add Jobs
Remove Jobs
Add Scholarships
Remove Scholarships
Add User/Professor
Remove User/Professor
Logout
Admin
View Wishlisted/enrolled courses
Post/Reply in Discussions
Login
Have Wishlist
Add Courses to wishlist
<<exclude>>
Remove from Wishlist
Student
View Profile
Request Appointments
See Faculty list
<<exclude>>
Update profile
Have To-do List
Add tasks to to-do list
<<exclude>>
Delete Tasks
See approved Appointments list
<<exclude>>
See Scholarship List without deadlines
Clear tasks list
Make Payment
Basic
See Jobs List without deadline
Purchase course
Become Premium User
Premium
See Scholarship List with deadlines & contact
See Research Offers
See Jobs List with deadline & contact

Software Testing

The software testing mechanism consists of both manual and automated testing. We have emphasized on the automated testing for our project, namely "*unit testing*" where we took a single component of our application and kept testing it until we achieved perfection.

8.1 Unit Testing

Unit testing falls under the white box testing methodology. Since it covers the unit or a small piece of code under test, thus its name. This was done by preparing some **test cases** and running **test statements** in order to compare the **actual results** with the **expected results**. Some tested components are listed below:

Component: User Registration

Test Case: Verify that new user can sign in successfully
Test Method: Try inserting valid email and user name, password and press the submit button
Actual Result: New User account is created in the database
Expected Result: New User account is created in the database and the user can now log in

Test Case: Verify that sign in fails with incomplete user data such as email
Test Method: Try inserting all information except email and press the submit button
Actual Result: No entry is created in the database
Expected Result: No new User account is created in the database

Component: Discussion Posting

Test Case: Verify that a student can post in discussion
Test Method: Write text in prompt and press the post button
Actual Result: The new discussion is posted in a thread
Expected Result: The new discussion is posted in a thread which can be seen by the one who posted and every other student registered.

Test Case: Verify that posting fails if prompt is empty
Test Method: Try keeping the field blank and hitting the post button
Actual Result: Nothing is posted
Expected Result: Nothing is posted

Component: Socket.IO Communication

Test Case: Verify that a professor can send a message to the chatroom

Test Method: Professor sends a message in the chatroom.
Actual Result: The message is sent and displayed in the chatroom.
Expected Result: The message is sent and displayed in the chatroom for all participants, including the professor.

Test Case: Verify that a student can send a message to the chatroom
Test Method: Student sends a message in the chatroom.
Actual Result: The message is sent and displayed in the chatroom.
Expected Result: The message is sent and displayed in the chatroom for all participants, including the student who sent it.

Component: Chatroom Creation and Joining

Test Case: Verify that a professor can create a chatroom
Test Method: Professor creates a chatroom.
Actual Result: A new chatroom is created with a unique code.
Expected Result: A new chatroom is created with a unique code, and the professor is added as a participant.

Test Case: Verify that a student can join a chatroom with a valid code
Test Method: Student enters a valid chatroom code and joins.
Actual Result: The student joins the chatroom successfully.
Expected Result: The student joins the chatroom successfully and can participate in live chatting.

Test Case: Verify that a student cannot join a chatroom with an invalid code
Test Method: Student enters an invalid chatroom code and attempts to join.
Actual Result: The student cannot join and receives an error message.
Expected Result: The student cannot join and receives an error message indicating the code is invalid.

Component: Performance and Load Testing

Test Case: Verify performance under load
Test Method: Simulate a large number of concurrent users sending messages.
Actual Result: System handles the load efficiently without significant degradation.
Expected Result: System handles the load efficiently without significant degradation in performance.

Test Case: Verify scalability across multiple servers .
Test Method: Deploy the application across multiple servers and test synchronization.
Actual Result: Data and messages are synchronized across servers.
Expected Result: Data and messages are synchronized across servers, maintaining consistency in the chatroom.

Component: Error Handling

Test Case: Verify error handling for invalid chatroom code
Test Method: Attempt to join a chatroom with an incorrect or expired code.
Actual Result: Error message is displayed.
Expected Result: Error message is displayed, guiding the user to enter a valid code.

Codes and Database

Last but not least, the codes and implementation of what went inside to make this entire website are listed below:

9.1 Frontend

We have used HTML, CSS, and JAVASCRIPT in the frontend Development. One of the main reasons to use HTML in frontend was to structure the content of the page seamlessly. It involves creating a hierarchy of components to represent titles, paragraphs, images, links, forms, and other content. We used CSS mainly to apply styles to the HTML (how they should look). It included a variety of colors, fonts, margins, paddings etc. to choose from. Lastly, we used JAVASCRIPT which adds interactivity and dynamic behavior to web pages and allows us to create features like interactive forms, animations, real-time updates, and more.

```
<!-- Javascript files-->
<script src="{{ url_for('static', filename= 'js/jquery.min.js') }}"></script>
<script src="{{ url_for('static', filename= 'js/popper.min.js') }}"></script>
<script src="{{ url_for('static', filename= 'js/bootstrap.bundle.min.js') }}"></script>
  <script src="{{ url_for('static', filename= 'js/jquery-3.0.0.min.js') }}"></script>
  <script src="{{ url_for('static', filename= 'js/plugin.js') }}"></script>
  <!-- sidebar -->
  <script src="{{ url_for('static', filename= 'js/jquery.mCustomScrollbar.concat.min.js') }}"></script>
  <script src="{{ url_for('static', filename= 'js/custom.js') }}"></script>
  <!-- javascript -->
  <script src="{{ url_for('static', filename= 'js/owl.carousel.js') }}"></script>
  <script src="https:cdnjs.cloudflare.com/ajax/libs/fancybox/2.1.5/jquery.fancybox.min.js"></script>
  <script>
  $(document).ready(function(){
  $(".fancybox").fancybox({
     openEffect: "none",
     closeEffect: "none"
     });
     </script>
```

```
  <!-- banner section start-->
  <div class="banner_section layout_padding">
      <div class="container">
          <h1 class="best_taital">The Best Grooming Tool for Graduates</h1>
          <div class="box_main">
              <input type="" class="email_bt" placeholder="Search Job" name="">
              <button class="subscribe_bt"><a href="#">Search</a></button>
          </div>
          <p class="there_text">There are many variations of passages of Lorem Ipsum available, but the majority have suffered alterationThere are many vari
          <div class="bt_main">
              <div class="discover_bt"><a href="contact.html">Sign up Today</a></div>
          </div>
      </div>
  </div>
  <!-- banner section end-->
```

```
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Communication Platform</title>
<style>
  body {
    font-family: Arial, sans-serif;
    line-height: 1.6;
    margin: 0;
    padding: 0;
    background-color: #e0d3ed;
  }

  .innercard {
    max-width: 800px;
    margin: 0 auto;
    padding: 20px;
  }

  .question {
    border: 1px solid #ddd;
    background-color: #fff;
    padding: 20px;
    margin-bottom: 20px;
    border-radius: 5px;
    box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);
```

```
      border-radius: 5px;
      box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);
    }

    .question h2 {
      font-size: 24px;
      margin: 0 15 0px;
      color: #007BFF;
    }

    .question p {
      margin: 0;
      color: #444;
    }

    .replies {
      margin-top: 5px;
    }

    .reply {
      border: 1px solid #ddd;
      background-color: #f9f9f9;
      padding: 15px;
      margin-bottom: 2px;
      border-radius: 5px;
    }

    .q {
```

9.2 Backend

The backend is capable of handling demands from the frontend, executing the essential logics, and returning suitable responses. In our backend we have used the **Flask framework**. Flask permits building web applications by characterizing routes, dealing with requests and responses, and association with databases. Here, we have used extensions for connecting databases like Flask-SQLAlchemy, creating APIs, ensuring security with authentication, and managing data processing and communication between the frontend and databases. We also used PHPmyadmin to create our database and store those data there.

USER CLASS:

User class is an abstract class where two members (student, professor) and admin class will extend this class. Necessary front-end codes are connected through the include statement.

Functions	Descriptions
Search()	User (students) can search for Jobs, User(Professor) can search for students through this function.
Post()	User (Professors) can post if they need any research assistant through this function.
Request()	User (students) can request for thesis meeting through this function.
payment()	User can purchase courses by payment through this function.
discussion()	Users can discuss if they face any problem.
delete()	User(professor) can delete requests and the admin class can also remove any type of user.

MEMBER CLASS:

Functions	Descriptions
Sign_in()	Users need to sign in before login.
Post()	User (Professors) can post if they need any research assistant.
Update()	Users can update their profile.
Request()	User (students) can request for thesis meeting.
approve()	User (Professor) can approve the request.
payment()	User can purchase courses by payment.
discussion()	Users can discuss if they face any problem.
Update()	Users can update their profile.
To_do()	User can bookmark their desired courses and create a wishlist.
delete()	User(professor) can delete requests.
sign_out()	User ends its connection by signing out.

```
class Contacts(db.Model, UserMixin):
    serialno = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), nullable=False)
    email = db.Column(db.String(20), nullable=False)
    password = db.Column(db.String(12), nullable=False)
    basic_mode = db.Column(db.Boolean, nullable=False, default=True)
    image = db.Column(db.String(80), nullable=False)


    def get_id(self):
        return str(self.serialno)

class Profcontacts(db.Model, UserMixin):
    serial = db.Column(db.Integer, primary_key=True)
    FullName = db.Column(db.String(100), nullable=False)
    uniName = db.Column(db.String(100), nullable=False)
    designation = db.Column(db.String(100), nullable=False)
    mail = db.Column(db.String(120), unique=True, nullable=False)
    passw = db.Column(db.String(100), nullable=False)
    status = db.Column(db.String(100), nullable=False)
```

ADMIN CLASS:

Admin class is another child class of user class. Front-end codes are connected through the include statement.

```
class Admin (db.Model, UserMixin):
    a_id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(80), nullable=False)
    email = db.Column(db.String(20), nullable=False)
    apass = db.Column(db.String(12), nullable=False)
    def get_id(self):
        return str(self.a_id)
```

Functions	Descriptions
log_in()	Admin needs to login before going to it's dashboard.
sign_out()	Admin can end their session with this function and close this website.
userl()	Admin can see through this function total user list of this website
userPl()	Admin can see through this function total user(professor) list of this website
remove()	Admin can remove any type of user through this function.

UPDATE METHOD:

```
@app.route('/update/>',methods=['POST','GET'])
def update():

    if(request.method=='POST'):
        name = request.form['name']
        email = request.form['email']
        password = request.form['password']
        current_user.name = name
        current_user.password= password
        current_user.email = email

        db.session.commit()
        flash("updated")
        return render_template("update.html")
```

Mainly, here we are using this function to update a user's current profile. At first here, we are getting the current values of the current user such as name,mail etc by querying the Contacts class. Then we are updating the current user and committing in the database.

DISCUSSION:

```
class Query(db.Model):
    q_id= db.Column(db.Integer, primary_key=True)
    question= db.Column(db.String(250),nullable=False)
    date = db.Column(db.DateTime, default=datetime.datetime.utcnow)
    poster = db.Column(db.String(25), nullable=False)

class Reply(db.Model):
    r_id= db.Column(db.Integer, nullable=False)
    answer = db.Column(db.String(250), nullable=False)
    time = db.Column(db.DateTime, default=datetime.datetime.utcnow)
    ans_id = db.Column(db.Integer, primary_key=True)
    person = db.Column(db.String(25), nullable=False)
```

```python
@app.route("/discussion", methods = ['GET', 'POST'])
def Discussion():
    ques = Query.query.filter().all()
    rep = Reply.query.filter().all()
    if (request.method == 'POST'):
        q = request.form.get('q')
        id = request.form.get('r_id')
        r = request.form.get('r')
        if q!=None and len(q) > 0:
            entry = Query(question=q, poster=current_user.name)
            db.session.add(entry)
            db.session.commit()
        elif r!=None and len(r) > 0:
            entry = Reply(r_id=id, answer=r, person=current_user.name)
            db.session.add(entry)
            db.session.commit()
        return redirect('/discussion')
    return render_template('discussion.html', ques=ques, reply=rep)
```

Mainly here, This Flask route "/discussion" handles GET and POST queries. It fetches all "Query" and "Reply" database entries. It examines POST requests for questions or replies, adds the corresponding record to the table using SQLAlchemy ORM, and redirects to the discussion page. Last, it renders the "discussion.html" template with the queries and replies as context variables.The route retrieves database questions and answers for discussion page requests.It monitors POST requests for new questions or replies, adds them to the table, and redirects to the same discussion page.

APPOINTMENTS:

```python
@app.route('/posts', methods=['POST', 'GET'])
def posts():
    if(request.method=='POST'):
        nm = request.form.get('nm')
        reason = request.form.get('reason')
        appointment = request.form.get('appointment')

        entry1 = Post(nm=nm, reason= reason, appointment=appointment )
        db.session.add(entry1)
        db.session.commit()
        flash("Sent appointment to professor")
        return render_template('posts.html')

    flash("Sorry couldn't Sent appointment to professor")
    return render_template('posts.html')
```

The Flask route "/posts" accepts POST and GET requests. When a POST request is received, it collects form data (name, reason, and appointment details), creates a new "Post" database item using SQLAlchemy, and displays a success flash message before rendering the "posts.html" template. It displays an error flash message and renders the same template for GET queries.

User input (name, reason, appointment) is added to the database's "Post" table for POST queries, and a success flash message is shown.
GET queries display an error flash message and "posts.html" template.
Flask, SQLAlchemy, flash messages, and HTML templates manage appointment request submissions and provide feedback based on request outcomes.

WISH_LIST:

```python
@app.route("/wishlist", methods=['GET', 'POST'])
def Wish_list():
    r = Cart.query.filter().all()
    wi = Wishlist.query.filter().all()
    pur = Purchase.query.filter().all()
    ids = []
    course_ids = []
    for i in wi: ids.append(i.user_id)
    for i in wi:
        if i.user_id == current_user.serialno:
            course_ids.append(i.wish_list)

    ids_pur = []
    course_ids_pur = []
    for i in pur: ids_pur.append(i.user)
    for i in pur:
        if i.user == current_user.serialno:
            course_ids_pur.append(i.course)

    if (request.method == 'POST'):
        q = request.form.get('type')
        w = request.form.get('wish')
        rem = request.form.get('remove')
        unen = request.form.get('unenroll')

        if rem != None:
            i,c=rem.split('-')
            entry_to_delete = Wishlist.query.filter_by(user_id=i, wish_list=c).first()
            db.session.delete(entry_to_delete)
            db.session.commit()
            return redirect('/wishlist')

        if w!=None and w not in course_ids:
            entry = Wishlist(user_id=current_user.serialno, wish_list=w)
            db.session.add(entry)
            db.session.commit()
            return redirect('/wishlist')

        if unen != None:
            i, c = unen.split('-')
            entry_to_delete = Purchase.query.filter_by(user=i, course=c).first()
            db.session.delete(entry_to_delete)
            db.session.commit()
            return redirect('/wishlist')

        if q != None:
            global course_taken
            course_taken = q
            return redirect('/payment_course')
    return render_template('wishlist.html', items=r, wish=ids, c = course_ids,pur=pur, ids_p=ids_pur,
                           course_ids_p=course_ids_pur)
```

The Flask route "/wishlist" manages user wishlists, cart items, and purchased courses, handling GET and POST queries.The code handles POST requests to add, remove, or unenroll from bought courses based on form submissions.The page displays user-specific data from the "Wishlist", "Cart", and "Purchase" database tables, including wishlist, cart, and paid courses. To ensure proper data manipulation, the code separates user IDs and course IDs to organize adding, removing, and unenrolling.

ADD PRODUCT:

```python
def add_product_to_cart():
    cursor = None
    conn = None  # Initialize conn outside the try block

    try:
        _quantity = int(request.form['quantity'])
        _code = request.form['code']

        # Validate the received values
        if _quantity and _code and request.method == 'POST':
            conn = db.connect()
            cursor = conn.cursor(pymysql.cursors.DictCursor)
            cursor.execute("SELECT * FROM cart WHERE code=%s", _code)
            row = cursor.fetchone()

            itemArray = {
                row['code']: {
                    'course_name': row['course_name'],
                    'course_code': row['coursecode'],
                    'quantity': _quantity,
                    'price': row['price'],
                    'image': row['image'],
                    'total_price': _quantity * row['price']
                }
            }

            all_total_price = 0
            all_total_quantity = 0

            session.modified = True
            if 'cart_item' in session:
                if row['course_code'] in session['cart_item']:
                    for key, value in session['cart_item'].items():
                        if row['course_code'] == key:
                            old_quantity = session['cart_item'][key]['quantity']
                            total_quantity = old_quantity + _quantity
                            session['cart_item'][key]['quantity'] = total_quantity
                            session['cart_item'][key]['total_price'] = total_quantity * row['price']
                else:
                    session['cart_item'] = array_merge(session['cart_item'], itemArray)

                for key, value in session['cart_item'].items():
                    individual_quantity = int(session['cart_item'][key]['quantity'])
                    individual_price = float(session['cart_item'][key]['total_price'])
                    all_total_quantity += individual_quantity
                    all_total_price += individual_price
            else:
                session['cart_item'] = itemArray
                all_total_quantity += _quantity
                all_total_price += _quantity * row['price']

            session['all_total_quantity'] = all_total_quantity
            session['all_total_price'] = all_total_price
```

The "add_product_to_cart" function controls the process by which items are added to a shopping cart. It takes advantage of form data to determine the required amount and item code. After the data is verified, a connection is made to the database. Course information is gathered and organized into a "itemArray" dictionary. Existing cart items are used to compute cumulative totals for price and quantity. Quantities and pricing are adjusted if the shopping cart is reset. The cumulative totals are saved permanently to the session. When the user is finished adding products to the shopping cart, they are sent to the cart page using the url_for redirect method. The efficiency and adaptability of the cart view is greatly improved by this feature.

MODE (Premium or basics):

```
373
374 class Premium(db.Model):
375     c_id = db.Column(db.Integer, primary_key=True)
376     c_name = db.Column(db.String(250), nullable=False)
377     wishlist = db.Column(db.Enum(WishlistStatus), nullable=False, default=WishlistStatus.no)
378     cart = db.Column(db.Enum(CartStatus), nullable=False, default=CartStatus.no)
379
380 @app.route("/profile", methods=['GET', 'POST'])
381 def profile():
382     if (request.method == 'POST'):
383         q = request.form.get('type')
384         print(q)
385         if q == "True":
386             return redirect('/payment')
387         else:
388             current_user.basic_mode = True
389         db.session.commit()
390         return redirect('/profile')
391     return render_template('profile.html')
392
```

This Flask route, "/profile", accepts GET and POST queries. It verifies form data for 'type' in POST requests. If "True", leads to '/payment'. If not, it redirects to the '/profile' page and sets the user's 'basic_mode' property to True in the database. It renders "profile.html" for GET queries.

POST requests with 'type' "True" lead to the payment page. If not, it modifies the user's 'basic_mode' attribute and redirects to the profile page.
The "profile.html" template displays the user's profile for GET queries.
This code uses Flask, SQLAlchemy, and HTML templates to manage user profiles and mode settings, allowing users to choose between basic and advanced modes with payment redirection.

ENROLL COURSE:

```python

@app.route("/enroll_courses", methods=['GET', 'POST'])
def Enroll_courses():
    r = Cart.query.filter().all()
    wi = Wishlist.query.filter().all()
    pur = Purchase.query.filter().all()
    ids = []
    course_ids = []
    for i in wi: ids.append(i.user_id)
    for i in wi:
        if i.user_id == current_user.serialno:
            course_ids.append(i.wish_list)

    ids_pur = []
    course_ids_pur = []
    for i in pur: ids_pur.append(i.user)
    for i in pur:
        if i.user == current_user.serialno:
            course_ids_pur.append(i.course)
    if (request.method == 'POST'):
        w = request.form.get('wish')
        rem = request.form.get('remove')
        unen = request.form.get('unenroll')
        if rem != None:
            i, c = rem.split('-')
            entry_to_delete = Wishlist.query.filter_by(user_id=i, wish_list=c).f
            db.session.delete(entry_to_delete)
            db.session.commit()
            return redirect('/enroll_courses')
        if w != None and w not in course_ids:
            entry = Wishlist(user_id=current_user.serialno, wish_list=w)
            db.session.add(entry)
            db.session.commit()
            return redirect('/enroll_courses')
        if unen != None:
            i, c = unen.split('-')
            entry_to_delete = Purchase.query.filter_by(user=i, course=c).first()
            db.session.delete(entry_to_delete)
            db.session.commit()
            return redirect('/enroll_courses')

    return render_template('enroll_courses.html', items=r, wish=ids, c=course_ic
                           course_ids_p=course_ids_pur)
```

This Flask route, "/enroll_courses", manages course enrollment, wishlists, and purchases via GET and POST queries. It displays user-specific data from the "Cart", "Wishlist", and "Purchase" databases. It handles POST requests to add/remove wishlist items and unenroll from courses. Finally, it renders "enroll_courses.html" with course data and user interactions. Manages course enrollment, wishlists, and purchases via "/enroll_courses" GET and POST methods.Retrieves user-specific data from "Cart", "Wishlist", and "Purchase" to display on the

page. User actions in the form include adding/removing wishlist items and unenrolling from bought courses.

PAYMENT

```
@app.route("/payment", methods=['GET', 'POST'])
def payment():
    error = None
    if (request.method == 'POST'):
        num = request.form.get('num')
        my = request.form.get('my')
        cvc = request.form.get('cvc')
        name = request.form.get('name')

        #verify number
        num_flag = num.isdigit() and (len(num)==6)
        my_flag = False
        if ('/' in my):
            x,y = my.split("/")
            if len(x) == len(y) and len(x) == 2 and x.isdigit() and y.isdigit():
                my_flag = True
        cvc_flag  = cvc.isdigit() and (len(cvc)==3)
        if num_flag and cvc_flag and my_flag:
            current_user.basic_mode = False
            db.session.commit()
            return redirect('/profile')
        else:error=True
    return render_template('payment.html',error=error)
```

The "/payment" Flask route is a payment processing page. GET and POST requests are processed without a hitch. Credit card details supplied by a user are retrieved during POST processing, including the card number, expiration date, CVV, and cardholder name. After that, the code makes sure that the card number, expiration date, and CVC are valid by checking their formats and lengths. If the input data passes validation, the 'basic_mode' attribute of the user is changed from True to False in the database to reflect the upgraded mode. This takes the user to their profile page at the '/profile' URL. In the event that the check fails, a corresponding error flag will be set to True. The payment form and any relevant problem messages are displayed after the "payment.html" template has been rendered.

TO-DO LIST:

```python
@app.route('/todo')

def todo():
    t = Task.query.filter_by(user=current_user.serialno).all()
    if (request.method == 'POST'):
        task = request.form.get('newtask')
        date_ = request.form.get('time')
        post_s = request.form.get('serial')
        clr = request.form.get('cl')
        if clr!=None:
            entries_to_delete = Task.query.filter_by(user=current_user.serialno).all()
            for entry in entries_to_delete:
                db.session.delete(entry)
            db.session.commit()
            return redirect('/todo')
        if post_s!=None:
            entry_to_delete = Task.query.filter_by(id=post_s).first()
            db.session.delete(entry_to_delete)
            db.session.commit()
            return redirect('/todo')
        if date_!=None and task!=None:
            entry = Task(user=current_user.serialno, description=task, date=date_)
            db.session.add(entry)
            db.session.commit()
            return redirect('/todo')
    return render_template('todo.html', tasks=t)

@app.route('/update',methods=['POST','GET'])
```

A user's to-do list is handled by the "todo" command. The function, when called, retrieves tasks from the "Task" table that correspond to the serial number of the current user. In response to a POST request, the function handles user activity related to task management. The program gleans information about the tasks, such as the tasks' names, due dates, and unique IDs. Clearing the full list of tasks ("clr") triggers the method to track down all instances of the given task for the current user and remove them. Using the post identifier ("post_s") provided, we may also delete certain tasks. Newly added tasks need to have both a description and a deadline date. New assignments are recorded in the "Task" table and associated with the user's serial number. The user's to-do list is then displayed once the function renders the "todo.html" template. This feature enables users to efficiently engage with and manage their to-do lists by allowing for the addition, removal, and visualization of tasks via a web interface.

JOB LIST:

```python
@app.route("/new_jobs", methods = ['GET', 'POST'])
def new_jobs():
    try:
        if(request.method=='POST'):
            '''Add entry to the database'''
            # Job_Code = request.form.get('Job_Code')
            Job_Title = request.form.get('Job_Title')
            Company_Name = request.form.get('Company_Name')
            Requirement = request.form.get('Requirement')
            Location = request.form.get('Location')
            Deadline = request.form.get('Deadline')

            entry = Jobs(Job_Title=Job_Title, Company_Name=Company_Name, Requirement=Requirement, Location=Location, Deadline=Deadline)
            db.session.add(entry)
            db.session.commit()
        return render_template('new_jobs.html')
    except:
        return render_template('contact_error.html')
    return render_template("new_jobs.html")
```

Both GET and POST requests for creating new job postings can be processed using the "new_jobs" Flask route. The function parses the form data and pulls out information like the job title, company name, requirements, location, and deadline if the request is a POST. The data is subsequently used to populate a new row in the "Jobs" table. The new job listing is saved via the db.session.add() and db.session.commit() procedures. Following its execution, the function displays the "new_jobs.html" template, which contains the job submission form.

9.3 Database

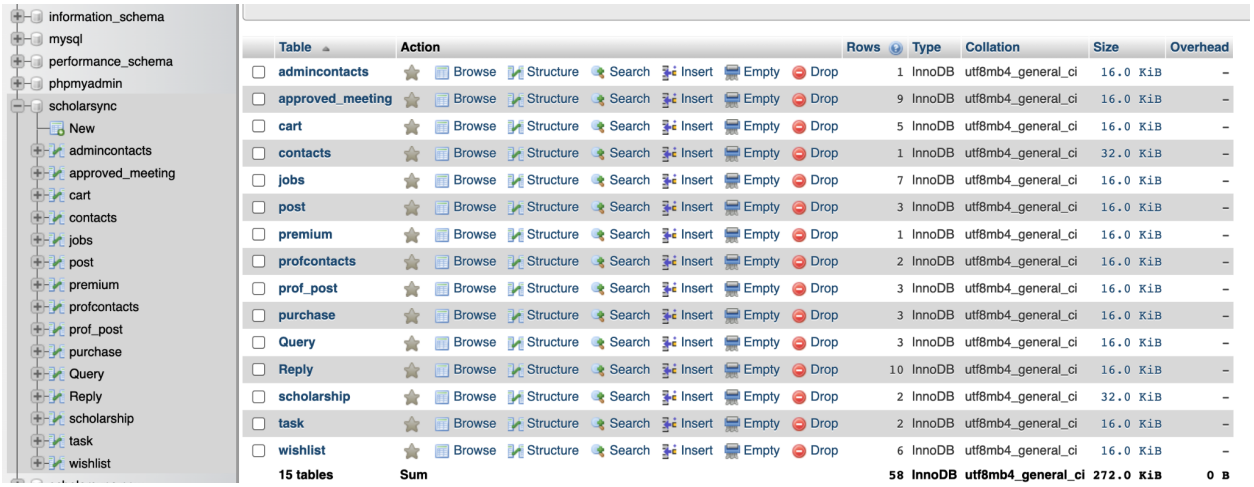


Table	Action	Rows	Type	Collation	Size	Overhead
admincontacts	Browse Structure Search Insert Empty Drop	1	InnoDB	utf8mb4_general_ci	16.0 KiB	-
approved_meeting	Browse Structure Search Insert Empty Drop	9	InnoDB	utf8mb4_general_ci	16.0 KiB	-
cart	Browse Structure Search Insert Empty Drop	5	InnoDB	utf8mb4_general_ci	16.0 KiB	-
contacts	Browse Structure Search Insert Empty Drop	1	InnoDB	utf8mb4_general_ci	32.0 KiB	-
jobs	Browse Structure Search Insert Empty Drop	7	InnoDB	utf8mb4_general_ci	16.0 KiB	-
post	Browse Structure Search Insert Empty Drop	3	InnoDB	utf8mb4_general_ci	16.0 KiB	-
premium	Browse Structure Search Insert Empty Drop	1	InnoDB	utf8mb4_general_ci	16.0 KiB	-
profcontacts	Browse Structure Search Insert Empty Drop	2	InnoDB	utf8mb4_general_ci	16.0 KiB	-
prof_post	Browse Structure Search Insert Empty Drop	3	InnoDB	utf8mb4_general_ci	16.0 KiB	-
purchase	Browse Structure Search Insert Empty Drop	3	InnoDB	utf8mb4_general_ci	16.0 KiB	-
Query	Browse Structure Search Insert Empty Drop	3	InnoDB	utf8mb4_general_ci	16.0 KiB	-
Reply	Browse Structure Search Insert Empty Drop	10	InnoDB	utf8mb4_general_ci	16.0 KiB	-
scholarship	Browse Structure Search Insert Empty Drop	2	InnoDB	utf8mb4_general_ci	32.0 KiB	-
task	Browse Structure Search Insert Empty Drop	2	InnoDB	utf8mb4_general_ci	16.0 KiB	-
wishlist	Browse Structure Search Insert Empty Drop	6	InnoDB	utf8mb4_general_ci	16.0 KiB	-
15 tables	Sum	58	InnoDB	utf8mb4_general_ci	272.0 KiB	0 B

The "Contacts" model saves user serial numbers, names, emails, passwords, basic mode status, and images.

The "Profcontacts" model shows professors' serial numbers, full names, universities, designations, emails, and passwords.

The "Admincontacts" model stores admins' serial numbers, names, emails, and passwords.

The "Jobs" model holds job listings with job code, title, firm name, requirements, location, and application deadline.

The "Scholarship" model includes scholarship code, name, university, requirements, country, application deadline, and link.

The "Query" model comprises user requests with IDs, questions, dates, and posters.

The "Reply" model saves query responses with ID, answer, time, and responder name.

The "Prof_post" model stores professor posts with ID, name, time, and content.

The "Cart" model manages user cart items with ID, course name, course code, image, and price.

The "Wishlist" model stores user wishlist items using unique IDs, user IDs, and desired courses.

The "Purchase" model stores course IDs, user IDs, and names.

The "Task" model records tasks with IDs, user IDs, descriptions, and dates.

User posts include ID, name, reason, and appointment data in the "Post" model.

The "ApprovedMeeting" model stores ID, name, purpose, and appointment details.

9.4 GitHub Repository Link

Link: https://github.com/TasniaJuha/CSE470-Project

Conclusion

In conclusion, the website's profound potential to shape the early careers of graduates is undeniable. By providing a comprehensive hub that caters to both students and professors, it creates a nexus of learning, networking, and skill enhancement. This platform paves the way for a new generation of professionals who are not only equipped with knowledge but also armed with the practical tools and connections necessary to flourish in an ever-changing world. Ultimately, it holds the promise of becoming an indispensable companion on the journey from academia to professional success, facilitating the transformation of passionate learners into accomplished contributors across various industries.