Indian Institute of Technology Gandhinagar

Team Data Sonic: Placement Management

CS 432: Databases

Assignment 4 : DEPLOYING THE DBMS

Aditi Dey (20110007)

Hetvi Patel (20110076)

Kamal Vaishnav (20110089)

Kevin Shah (20110096)

Lipika Rajpal (20110102)

Prakriti (20110142)

Preetam Chhimpa (20110145)

Rishab Jain (20110164)

Sagar Chavan (20110045)

Samiksha Kamble (20110182)

Taha Mohammad Syed (18110169)

INDEX

IMPORTANT NOTES

- Here are some of the user_id and passwords that can be used to log in:

	User ID	**Password**
Admin	29231375	password10
	29231375	password10
Company Representative	99490777	password10
	57188337	password10
Student	76876075	password10
	10013	Henry13196

NOTE: *All the same passwords are purely coincidental.*

- Github link to the repository containing final SQL dump

- Github link to the repository containing the code for the web application

2. Problem Statement & Requirements

1. All the relations and their constraints should be reflected from Assignment 1.
2. Login page with authentication of users and stakeholders.
3. The system should be safe from 4 attacks (2 on the front end and 2 on the back end), 1 additional attack by G1 or G2 will lead to 20 bonus points per G1/G2 subgroup.
4. Show the systems to respective stakeholders, and get feedback on how to improve and upgrade their systems.
5. Push it to GitHub and share the link for submission.

3. Tasks

3.1 Responsibility of G1: 40 Pts.

1. The G1 takes two feedbacks from the stakeholders, one initial feedback (on or before 30th March 11:59 PM), and then makes relevant changes as suggested per the first feedback, then final feedback (on or before 10th April 11:59 PM) post changes. The write-up/documentation should have screenshots before the first feedback, after the first feedback, and after the second feedback. If a team discusses with multiple stakeholders, please fill out the forms again (Initial and final feedback forms). (**30 Points**)

   First Feedback:

   Some of the changes that were made **after the first feedback** for our Placement Management System project was to improve user experience and add more features.

- For the **student stakeholder**, One of the key adjustments was the addition of an "Apply" button, enabling them to rapidly apply for job openings offered by companies, only in which they are eligible companies. The student portal had additional views to accommodate this functionality, enabling students to view the jobs they have applied for and the status of their application.

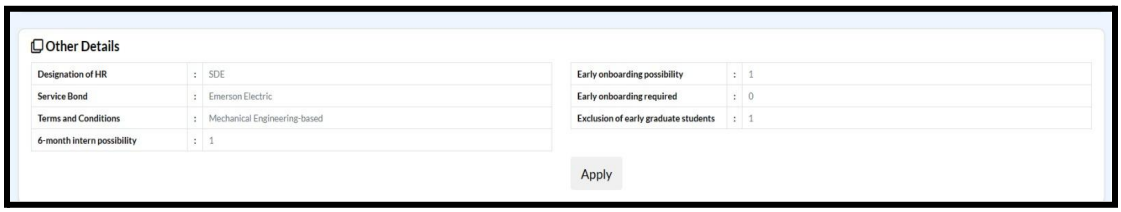


Fig.1 Option to Apply to Eligible Job

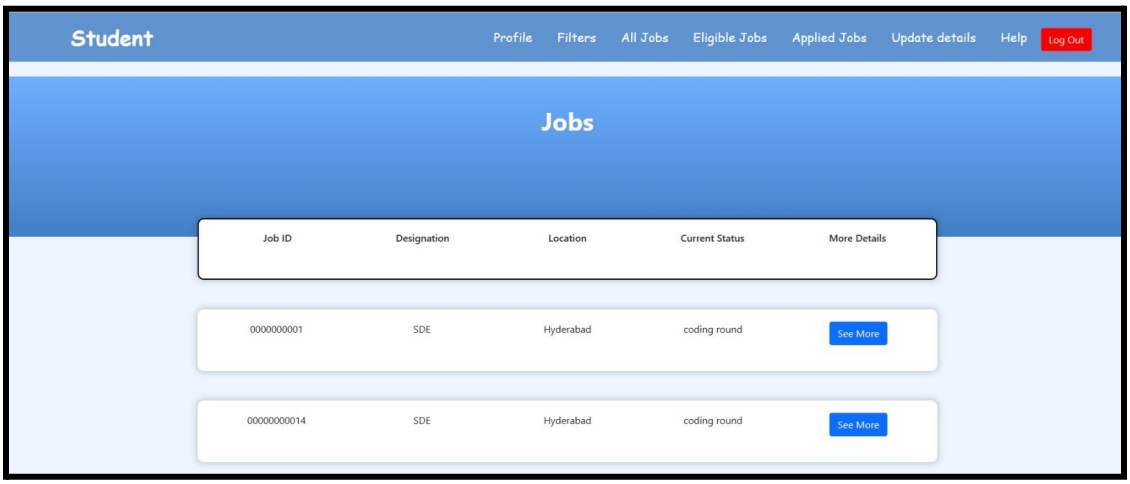


Fig.2 Tab to see the Job students have applied for

- For the **company stakeholder**, new views for Posted jobs have been added letting them view the jobs they have posted. This allows companies to more easily manage their job posts and monitor the status of their hiring process due to this functionality.

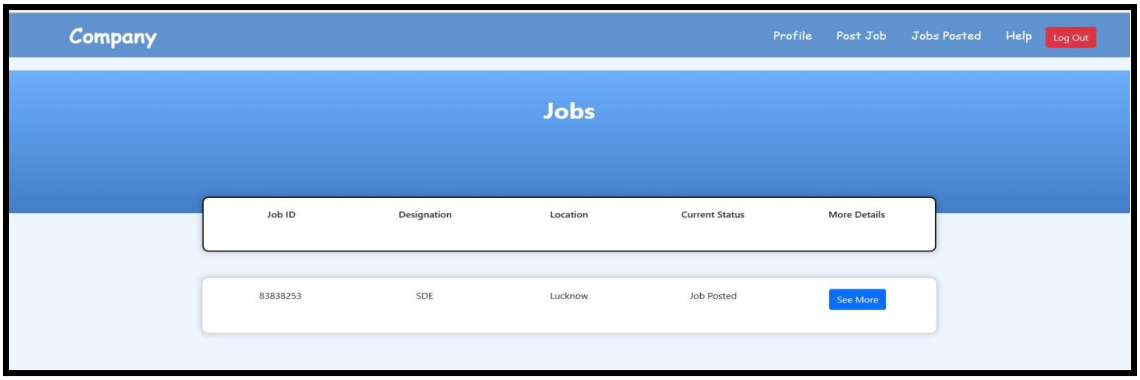


Fig.3 Tab to see the Jobs company representative have posted

- We made changes in the UI of the website to enhance the overall user experience. To offer a more user-friendly and unified appearance, the dashboards, registration, and login sections were modified. The login page was entirely changed in order to match the overall blue-white palette of the website.

Before **After**

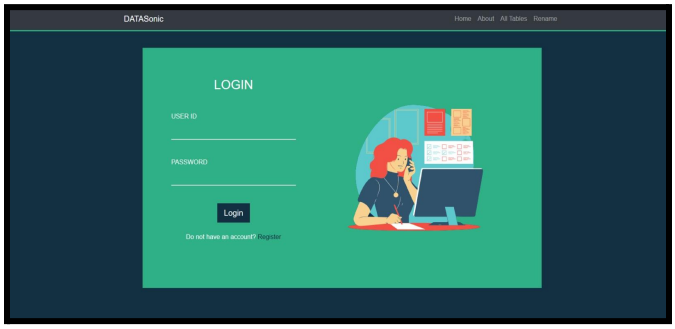


Fig.4

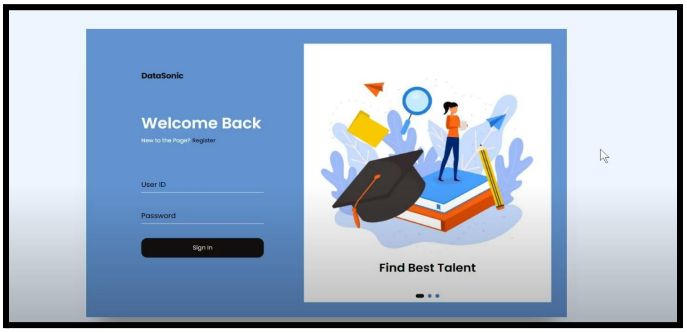


Fig.5

- More **Popups** have been made, now users can see a popup after every activity like insert, update, logging activity, etc.

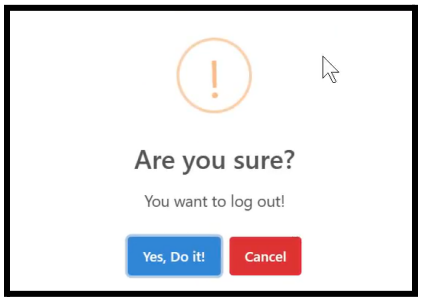


Fig.6

Second Feedback:

After the second feedback for the Placement Management System, further improvements were made to the student section of the platform. One of the fundamental changes was the addition of filter options to help students find job opportunities that match their interests and qualifications.

- The filter options added to the student profile include job_profile and job_category, allowing students to search results based on the disciples they are in or the type of opportunity like full-time, part-time, remote. These filters were designed to help students find job opportunities relevant to their career goals and interests.
  Please note that we have only added specific filters in this feedback phase, to show how filters can be implemented. Some other filters like salary etc could be added as per need.

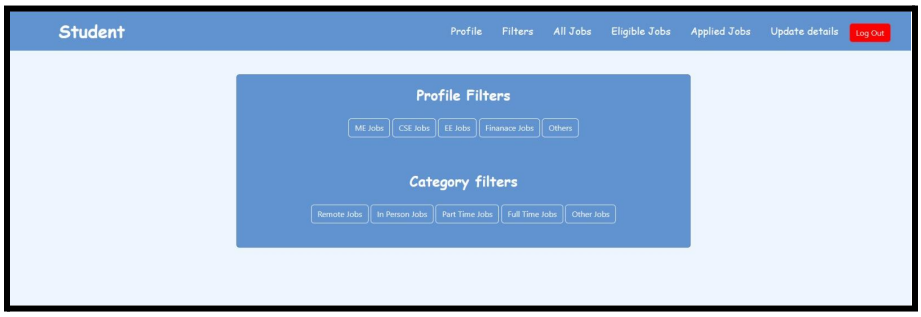


Fig.7

- A help option was added so that in case a student or company representatives faces difficulties they are able to write their message in it. These queries can be directly seen in the admin dashboard where all messages can be seen along with other details like email id, query, etc.

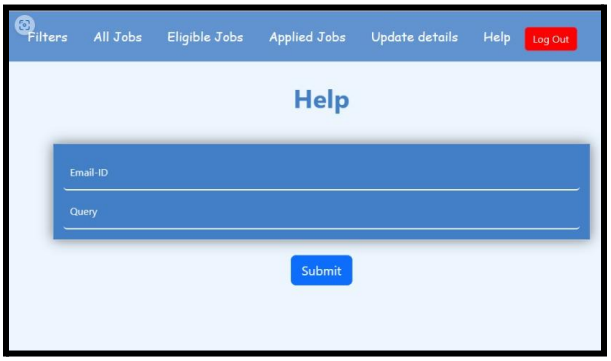


Fig.8

Fig.9

- To further improve the user experience, a Navbar was also added so that options are accessible easily which was earlier available only in the login page. This feature provides all the stakeholders with quick access to the different sections of the portal and makes it easier for them to navigate the platform.

Fig.10

Screenshots:

Before First Feedback:

Login Page

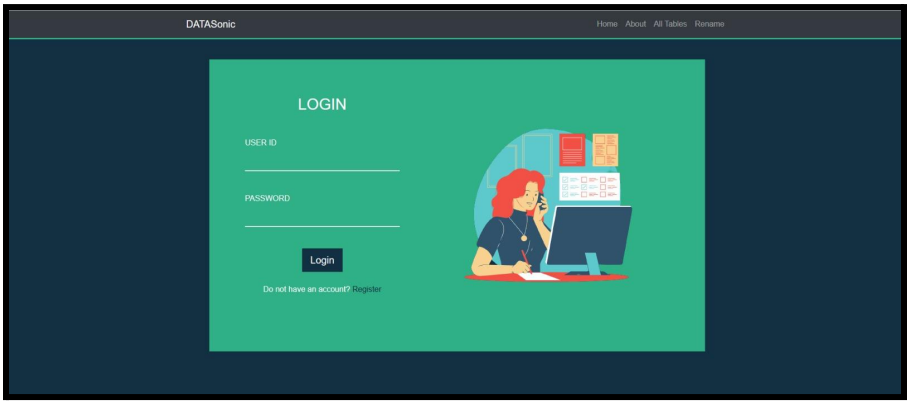


Fig.11

Registration Page:

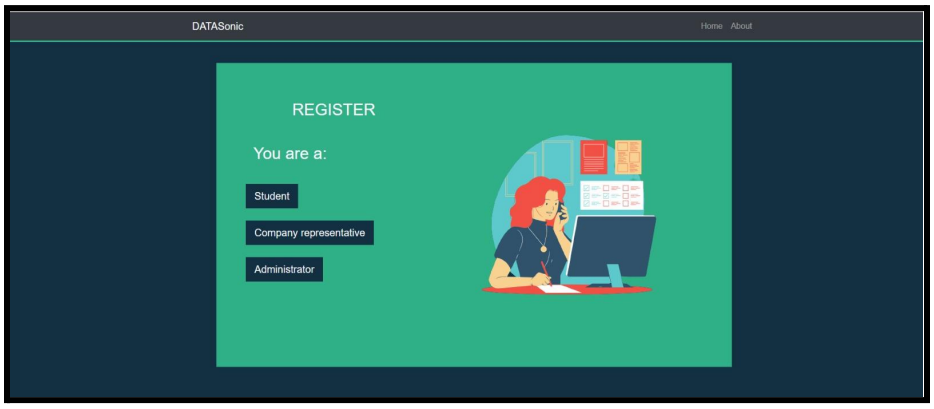


Fig.12

Student Dashboard

Fig.13

Admin Dashboard:

Fig.14

Company Representative Dashboard:

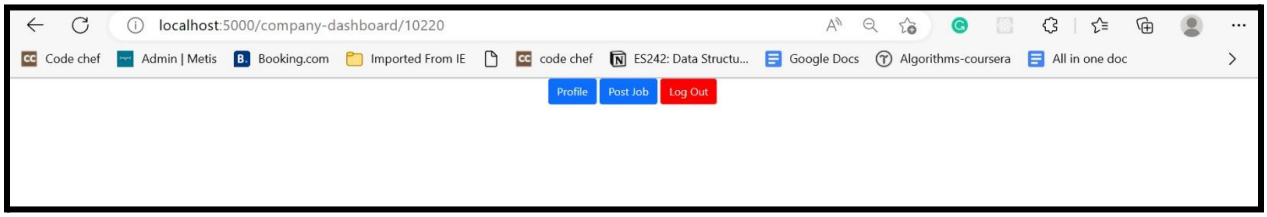


Fig.15

Student Registration Form:

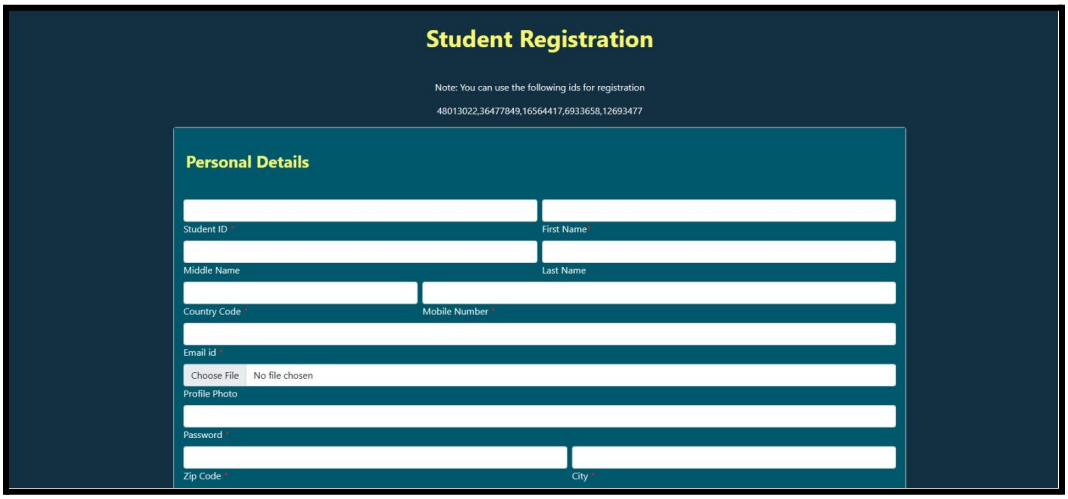


Fig.16

After First Feedback:

Login Page:

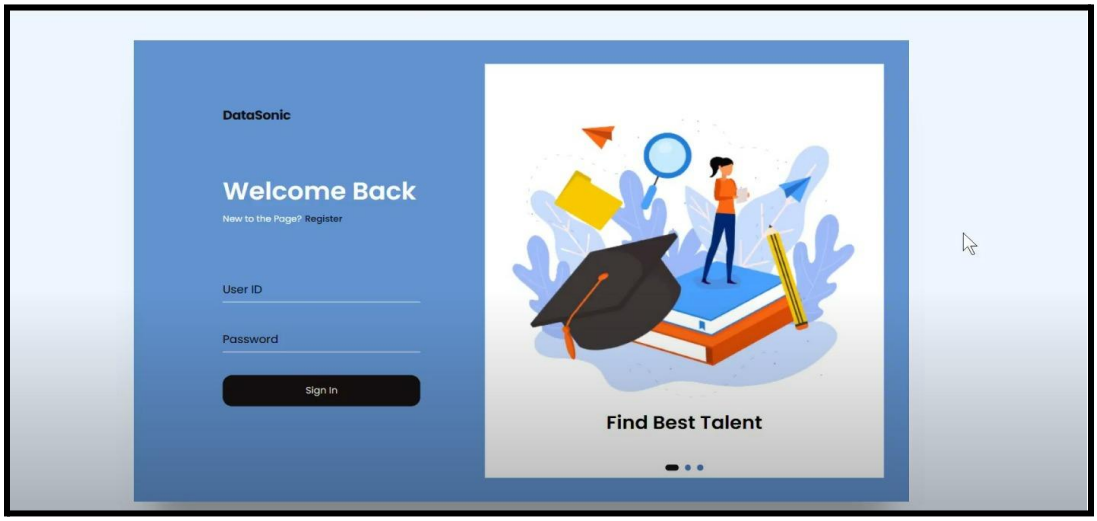


Fig.17

Registration Page:

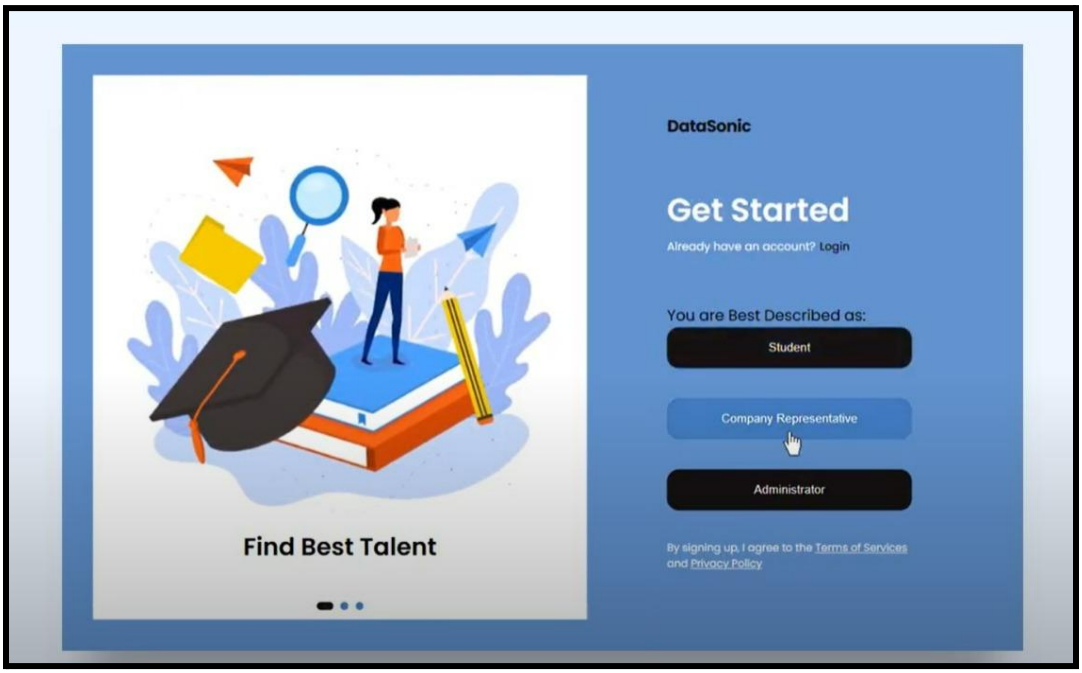


Fig.18

Student Dashboard:

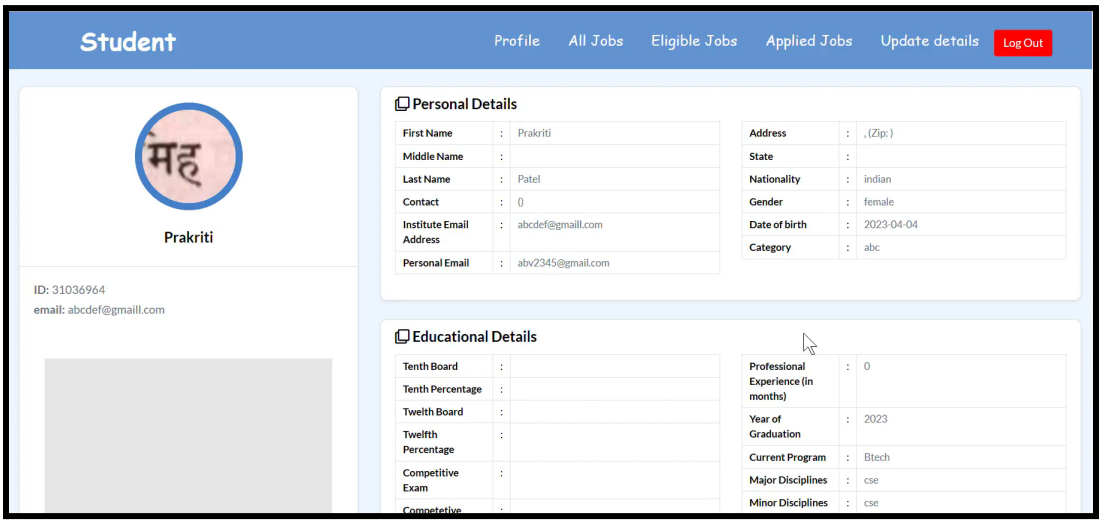


Fig.19

Admin Dashboard:

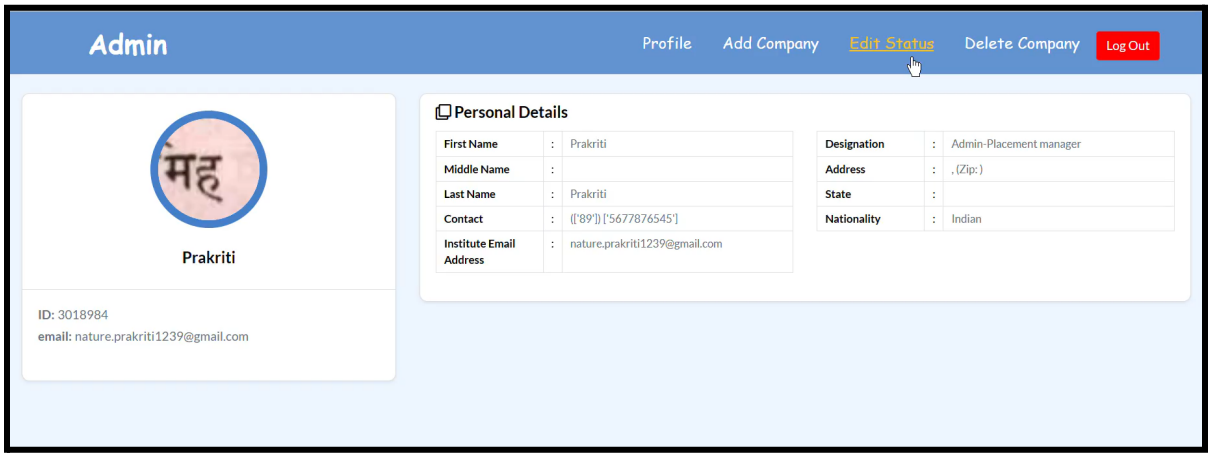


Fig.20

Company Representative Dashboard:

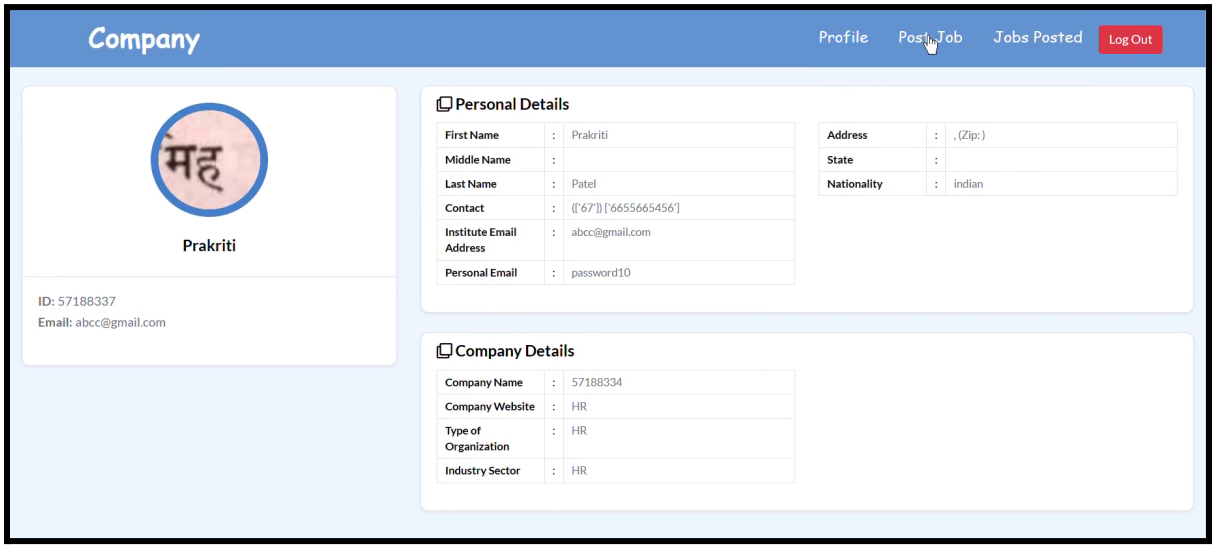


Fig.21

Student Registration Form:

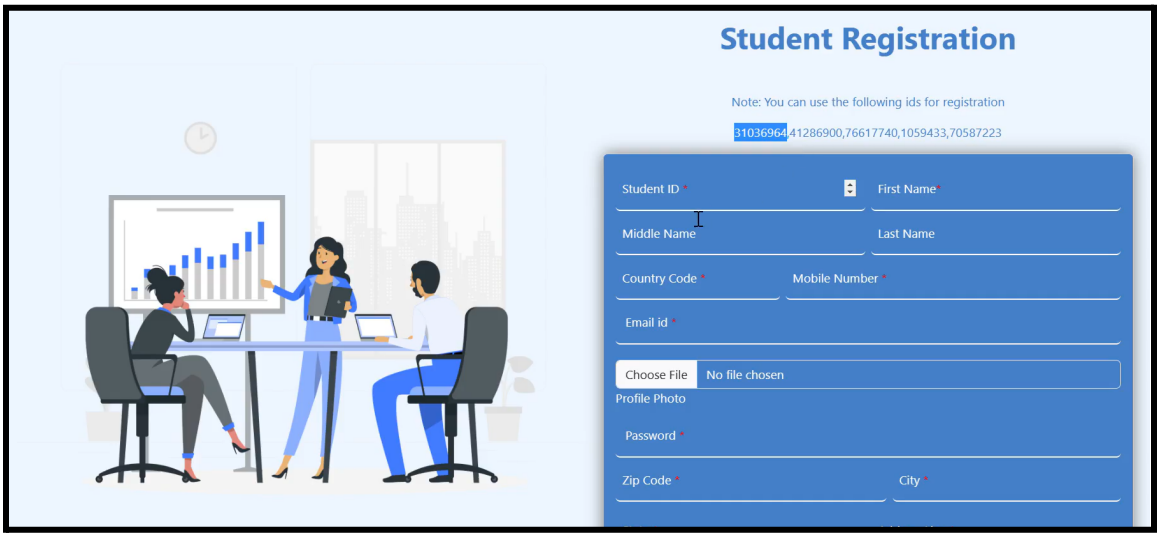


Fig.22

Videos:

Student: link

Admin: link

Company Representative: link

After Second Feedback:

Filter Option:

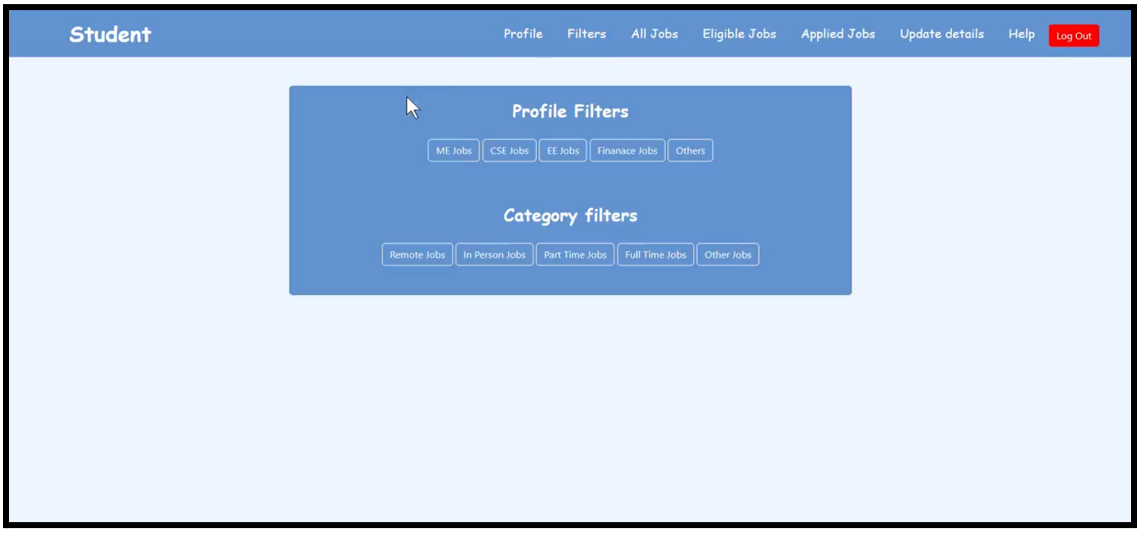


Fig.23

Help Option:

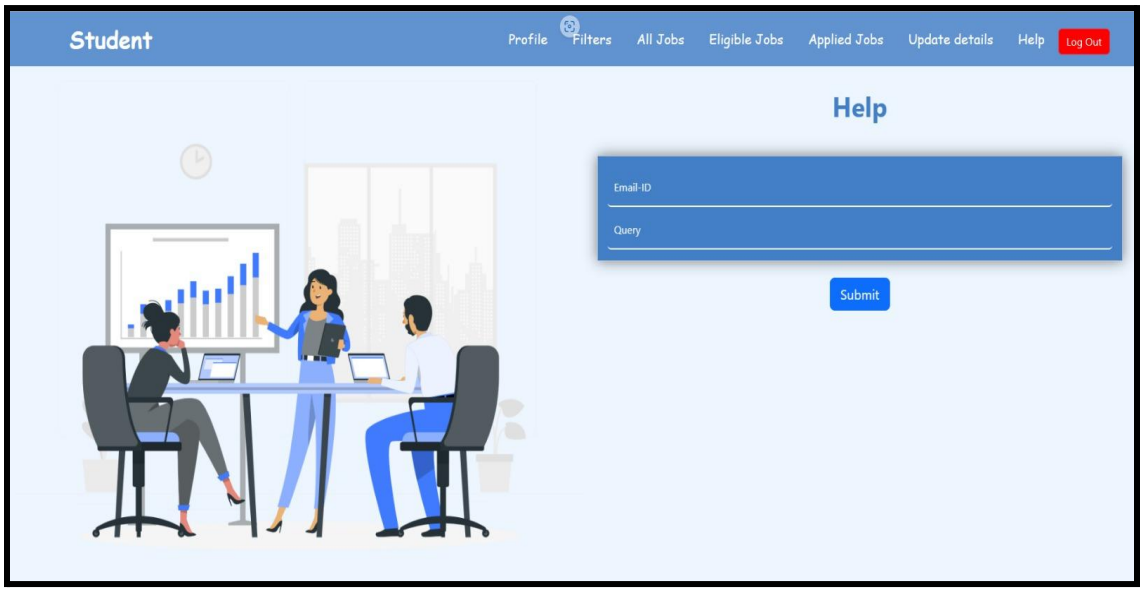


Fig.24

Navbar:

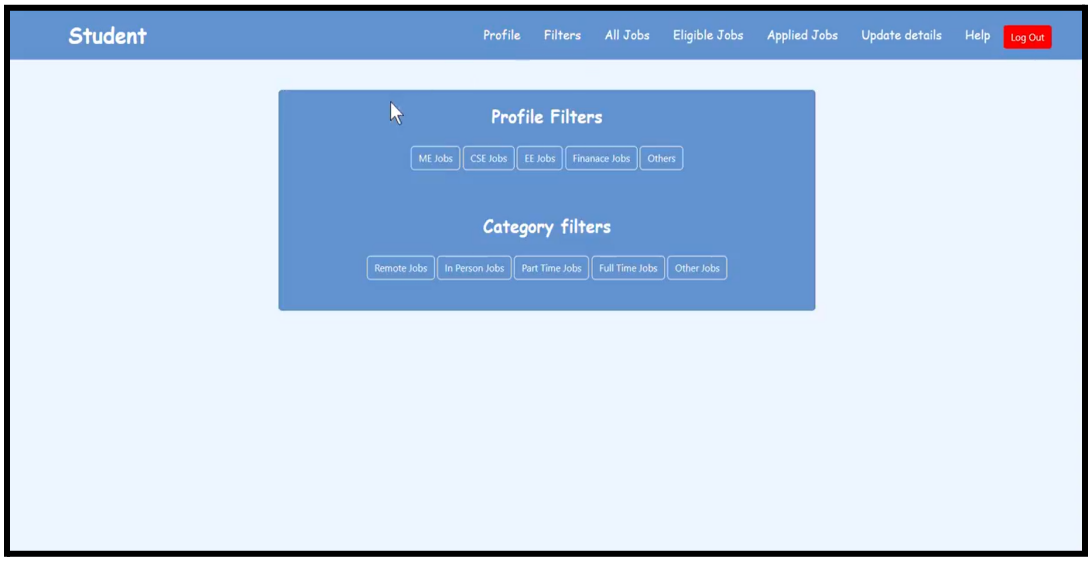


Fig.25

Videos:

Student : link

Admin: link

Company Representative: link

2. Attach screenshots of different views [along with a write-up on their privileges] of the database as seen by different classes of users. (**10 Points**)

Privileges to each users

1. Students :

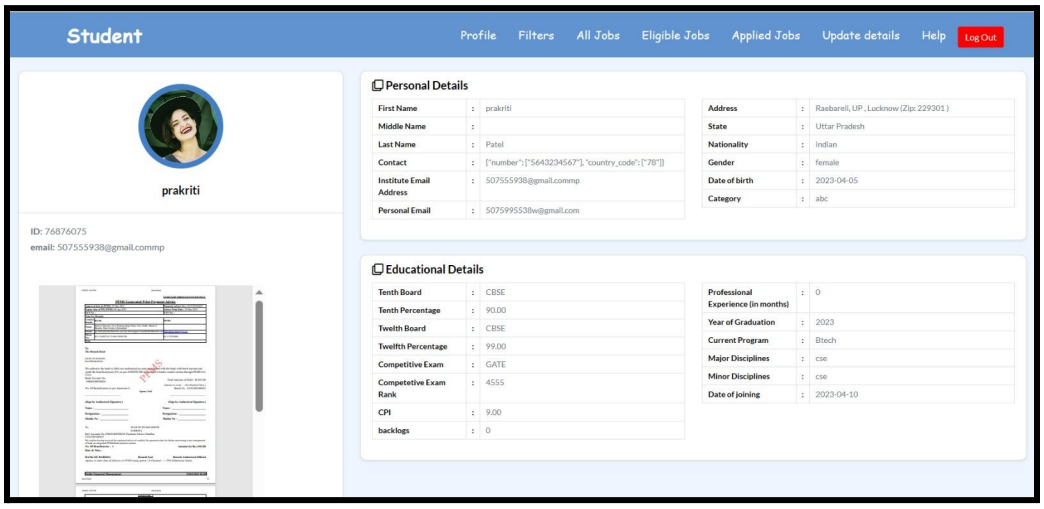


Fig.26

Privileges:

- Students can build and manage their own profiles on the portal, including personal and academic information. They can also change and update their profiles as they see fit.
- Students can see all the jobs that are posted in the all jobs option. They can also look at the job profiles that companies have posted, which include information such as the job title, description, qualifications, and salary range.
- They can then apply for jobs by uploading resumes and clicking the apply button for jobs they are eligible for. Apart from that they can also check the status of the jobs they have applied for in the Applied Jobs section.
- Students can use the filter button on navbar to filter the job posted according to the discipline they are in or the type of job they want to pursue like full-time, part-time,remote,etc.

2. Company Representative :

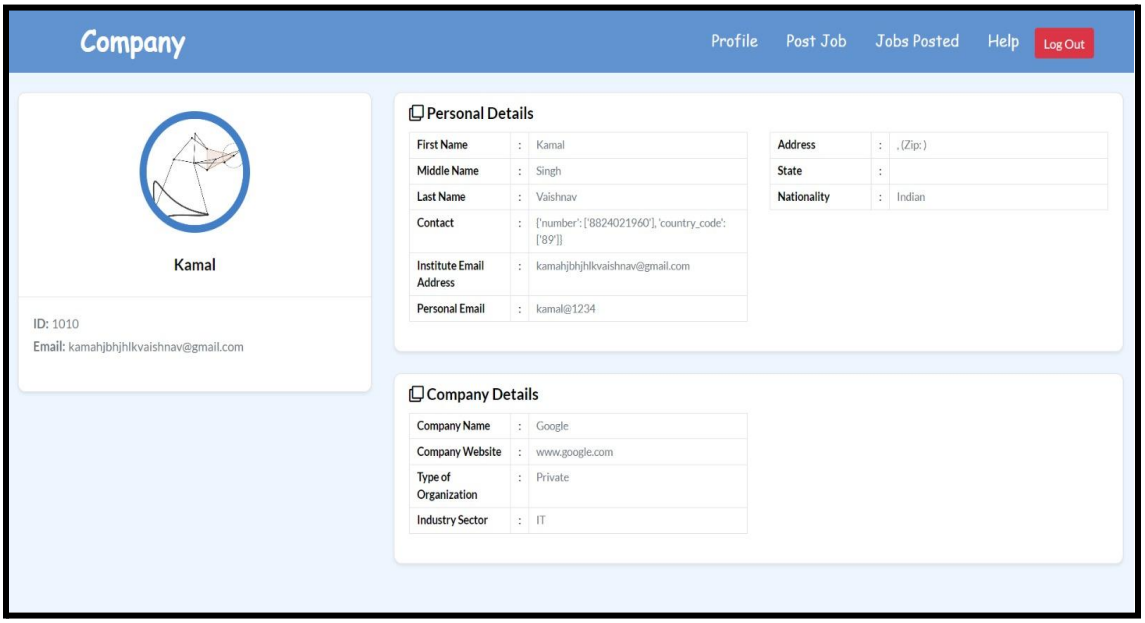


Fig.27

Privileges:

- Company representatives can build and manage their own profiles on the portal, including information such as their name, contact information, firm name, and job description.

- Company representative personnel can publish job openings on the site, including job title, description, qualifications, needed experience, and salary range.
- In case the company representative wants to see the job they have posted they can see it in the 'Jobs Posted' option.

Admin

Dashboard:

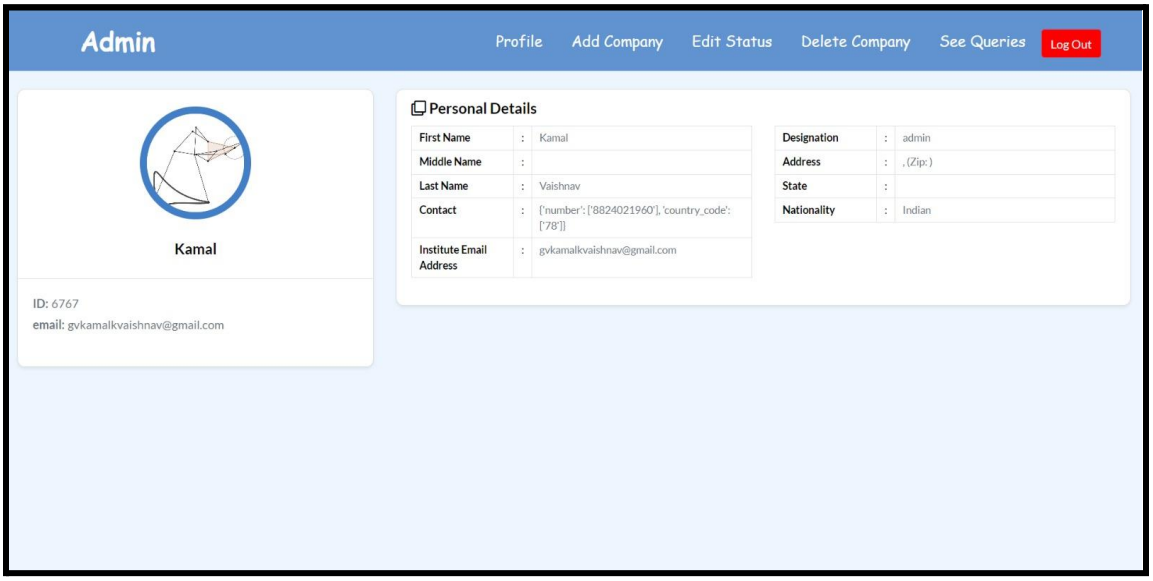


Fig.28

Privileges:

- The admin can build and manage their own profiles on the portal, including information such as their name, contact information, Institution Email Id, etc.
- The administrator can add company, delete company and their personal information can also be edited.
- Admin can see the queries posted by students and company representatives using the 'Help' tab in the Queries section, where they can see the information like Message, Email Id, etc .

3.2 Responsibility of G2: 40 Pts.

1. Concurrent multi-user access: Multiple users with different roles can access and update the database concurrently. In such a scenario, the same item can not be updated by two different users. For example, locks can be applied to tables in MySQL. (**10 Points**)

   This was the code before applying the locks:

```python
@app.route('/update-details/<person_id>', methods=['GET', 'POST'])
def update_cpi(person_id):
    if request.method == 'POST':
        userDetails = request.form
        newcpi = userDetails['cpi']
        experience = userDetails['experience']
        print("newcpi ", newcpi)
        print("experience ", experience)
        try:
            cur = mysql.connection.cursor()
            cur.execute("UPDATE student SET cpi = %s, professional_experience=%s WHERE person_id = %s", (
                        newcpi, experience, person_id))
            mysql.connection.commit()
            print("Data for cpi updated successfully")
            return redirect("/student-profile/"+str(person_id))

        except mysql.connection.Error as error:
            mysql.connection.rollback()
            error = "{}".format(error)
            return redirect("/student-profile/"+str(person_id))
    return render_template('dashboard/update_cpi.html', person_id=person_id)
```

   Fig.29

   When many transactions attempt to edit a table at the same time, the database may go into an inconsistent state that can lead to deadlock.
   To avoid this we can hold an exclusive lock on the table before updating and can release it after the update is done. By locking a table during an update, MySQL ensures that no other transaction can edit the same table at the same time, ensuring data consistency. Locking also aids in the prevention of deadlocks, which occur when several transactions are held indefinitely because each transaction is waiting for a resource that another process is holding.

   If two transactions are attempting to modify the CPI at the same time, and they have conflicting changes, they may end up waiting for each other to complete their operation, leading to a deadlock.

Hence we modified the code, such that when a student will be updating their CPI, an exclusive lock will be allocated to that transaction, and any attempts to update the CPI outside that session will not be considered.

After applying locks, the code appears as follows:

```
@app.route('/update-details/<person_id>', methods=['GET', 'POST'])
def update_cpi(person_id):
    if request.method == 'POST':
        userDetails = request.form
        newcpi = userDetails['cpi']
        experience = userDetails['experience']
        try:
            cur = mysql.connection.cursor()
            cur.execute("LOCK TABLES student WRITE")
            cur.execute("SELECT * FROM student WHERE person_id = %s", [person_id])
            record = cur.fetchone()
            if record:
                cur.execute("UPDATE student SET cpi = %s, professional_experience=%s WHERE person_id = %s", (
                        newcpi, experience, person_id))
                mysql.connection.commit()
                print("Data for cpi updated successfully")
                return redirect("/student-profile/"+str(person_id))
            else:
                error = "Data not found"
                return render_template("dashboard/error.html", person_id=person_id, error=error)
        except Exception as error:
            mysql.connection.rollback()
            error = "{}".format(error)
            return render_template("dashboard/error.html", person_id=person_id, error=error)

        finally:
        # Release the lock on the table
            cur.execute("UNLOCK TABLES")
            cur.close()
    return render_template('dashboard/update_cpi.html', person_id=person_id)
```

Fig.30

2. Implement the changes in the database as per the feedback received from stakeholders. (**30 Points**)

1) Filter option:

Answer:

We received feedback from a potential user as a 'student' that it would be convenient for the students to find suitable available jobs if we provided a filter on the job profiles and category(in person, remote, part-time, full-time, etc.). We have incorporated this feature in our web application.

Implementation:

For implementing the job filter(on profile and category), we created a new entity '*filters*'. The schema of this entity is shown in the following figure (Fig.31)

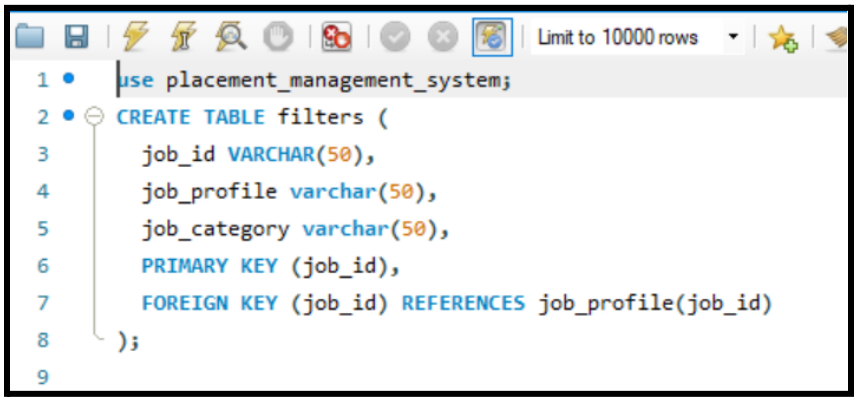


Fig .31 : New entity 'filters'

We had to create a new entity for filters instead of adding **job_profile** and **job_category** (see Fig above) as attributes to the entity '*job_profile*'. This is because otherwise we had to drop '*job_profile*' and some other tables (e.g. prog_details) and recreate them in order to resolve potential referential integrity constraints.

The following query takes the values of all the attributes of *filters* and enters them in a tuple in the table in the database →

```
sql = "INSERT INTO filters (job_id, job_profile, job_category) VALUES(%s,%s,%s)"
values = (job_id, job_profiles, job_categories)
cur.execute(sql, values)
print("Data for filters inserted successfully")
```

Integrity and key constraints:

It is logical to have only those entries/tuples in the entity '*filters*' for which a corresponding job entry exists in the entity '*job_profile*'(containing all the jobs). To ensure this, we have added a foreign key constraint in the schema of '*filters*'. The attribute **job_id** in *filters* is a foreign key that references the corresponding entry in *job_profile* with the same value of **job_id.** This ensures that we cannot enter a tuple in *filters* with a value of job_id that does not reference any entry in the entity job_profile.

We have verified that above constraints are executed successfully by running the following queries in our workbench:

- We used the INSERT INTO statement to insert values in the *filter* entity manually.
- We tried to enter two tuples for which a corresponding tuple existed in the job_profile, i.e., filters.job_id = job_profile.job_id [for corresponding tuples]. As shown in the following figure (Fig.32), these two tuples were inserted successfully.

Query 1:

```
INSERT INTO filters (job_id, job_profile, job_category) VALUES
("0000000002", "Other", "Remote");
```

Query 2:

```
INSERT INTO filters (job_id, job_profile, job_category) VALUES
("0000000012", "Other", "Remote");
```

- Then we tried to enter a tuple in *filters* that had a value of job_id attribute that did not match with any entry in the job_profile table. As expected, this is an invalid entry as it breaches the foreign key constraint. Hence, the insertion failed and threw an error (see fig 32).

Query 3:

```
INSERT INTO filters (job_id, job_profile, job_category) VALUES
("001", "Other", "Remote");
```

✓	2	22:45:25	use placement_management_system	0 row(s) affected	0.000 sec
✓	3	22:45:29	INSERT INTO filters (job_id, job_profile, job_category) VALUES ("0000000002", "...	1 row(s) affected	0.047 sec
✓	4	22:46:07	INSERT INTO filters (job_id, job_profile, job_category) VALUES ("0000000012", "...	1 row(s) affected	0.000 sec
✗	5	22:46:22	INSERT INTO filters (job_id, job_profile, job_category) VALUES ("001", "Other", "...	Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails (`...	0.015 sec

Fig.32

Functioning of the feature:

This feature is available in the **student** view of the web application. The user can view the filter options by clicking on the 'filters' button on the navigation bar (see Fig 33).

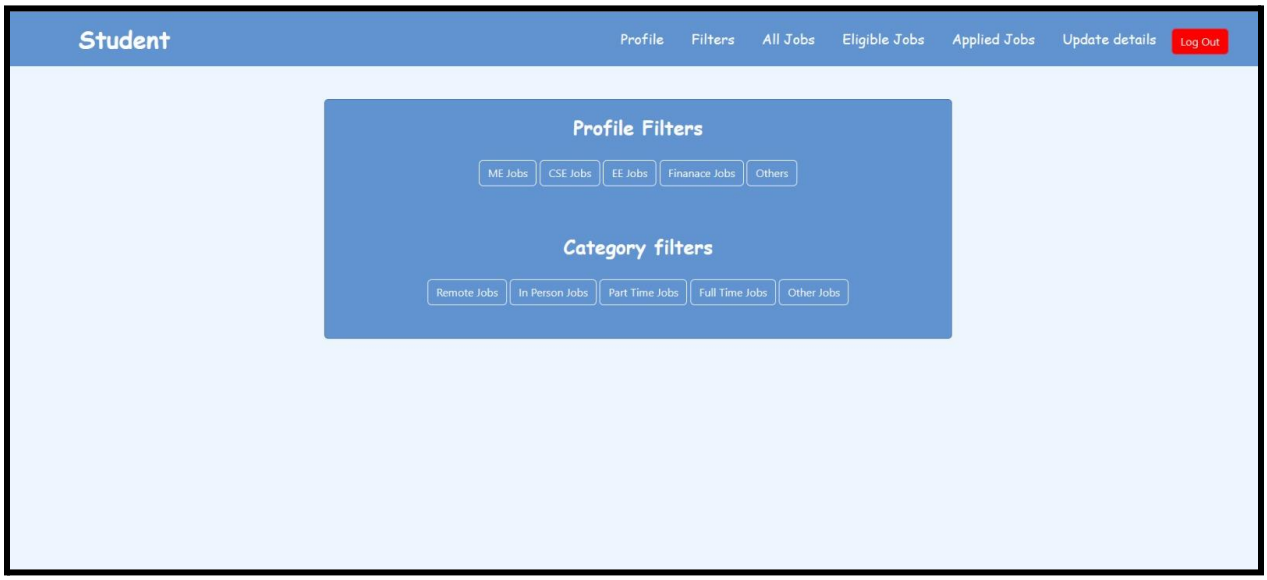


Fig.33 The filter page

In the above figure(Fig 33), we can see the page that presents the filtering options to the students on the basis of job profile and job category. We have included the following filters →

- **Job Profile:** ME Jobs, CSE Jobs, EE Jobs, Finance Jobs, Others.
- **Job categories:** Remote jobs, In person jobs, Part-time jobs, Full-time jobs, Others.

Note: *More options can be added in either filter by following a similar implementation.*

On the page shown in fig , a user can click on any option to view a list of all available jobs that fulfil the criteria in the option. For example, if a user clicks on the 'ME Jobs' in the 'Profile Filters', he/she will be redirected to a page showing the lists of all jobs related to Mechanical Engineering in a tabular form (see fig 34).

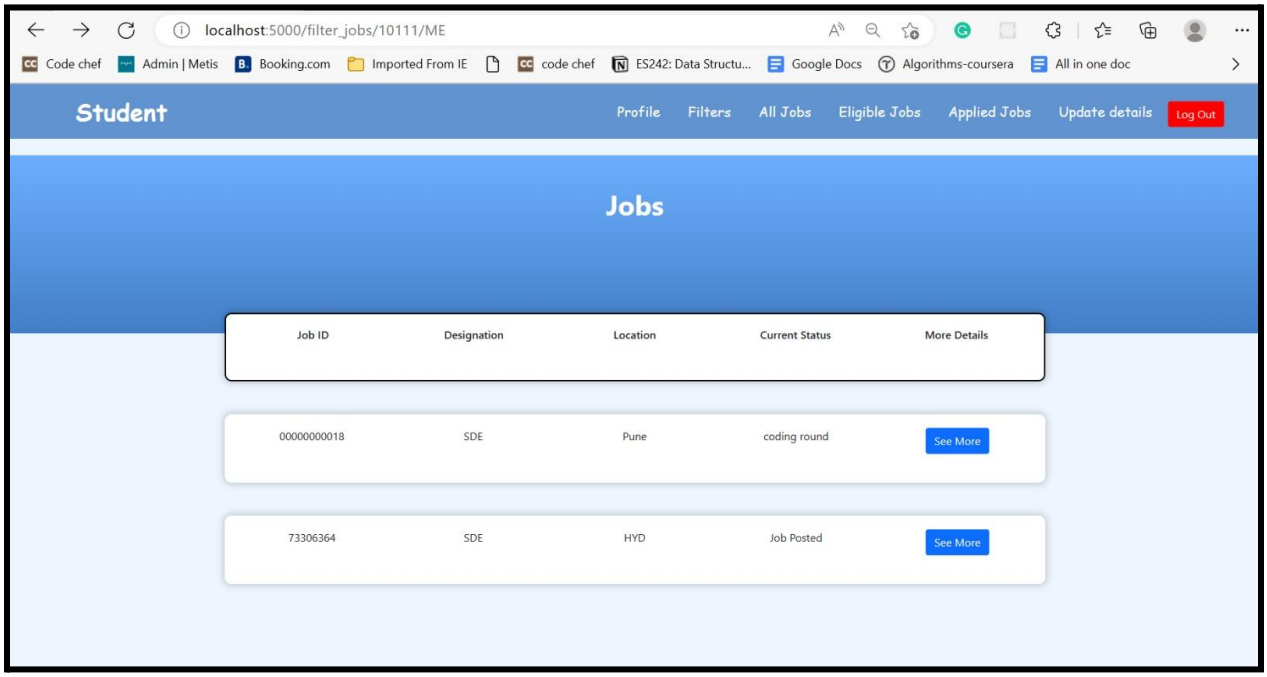


Fig 34: All jobs according to the chosen filter

In the above figure, the user can view details like designation, location, etc. For further details, he/she can click on the 'See More' button. This will redirect the user to the 'basic details' page of that particular job(see Fig 35).

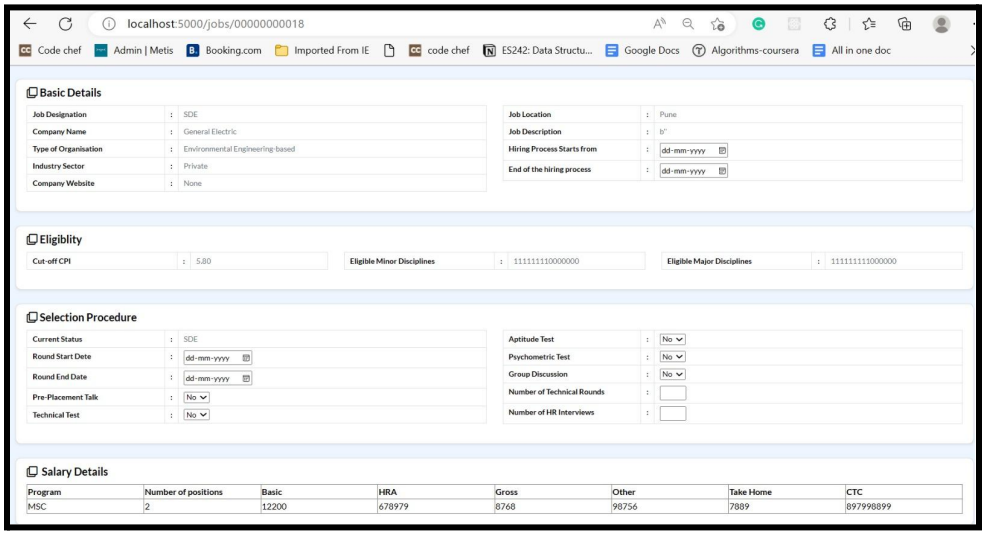


Fig 35: Details of the selected job

2] Help Option

We received feedback from one of our stakeholders - Pupul Singh Dhalbera (student) to include a help option where students will be able to ask queries to the admin during the process of placement.

Implementation

For implementing the help feature, we created a new entity '*queries*'. The schema of *queries* is shown below:

```
use placement_management_system;
CREATE TABLE queries (
  query_id INT AUTO_INCREMENT PRIMARY KEY,
  person_role VARCHAR(50),
  person_id int,
  message varchar(50),
  email_id varchar(50) UNIQUE,
  query_date varchar(50),
  FOREIGN KEY (person_id) REFERENCES person(person_id)
);
```

Fig.36

Attributes of the schema:

- *query_id* is the primary key of the entity
- *person_role* is entered by the person (student or company representative) while entering the query
- *person _id* is the id of the person (student or company representative) entering the question
- *message* is the question that the person asks
- *email_id* is the email_id of the person
- *query_date* is the date of sending the query

Referential integrity with the entity person is maintained as seen in the schema. A complete new entity queries has been created since referential integrity with the person entity has to be

maintained. person entity is related to various entities and a lot of tables will have to be dropped if queries are added as an attribute.

Consider the following query →

```
INSERT INTO queries (person_role, person_id, message, email_id, query_date)
VALUES ("company_rep", 10098, "this is a yet another message",
        "prakkriti.saroj@iitgn.ac.in2", "15-04-2023");
```

This query inserts the details of the person entering the question and adds the query to the table in the database. This query is sent to the admin.

Integrity and key constraints:

To maintain the consistency of the database foreign key constraint is added to the person_id attribute which refers to the person_id in the person table. This ensures that a person not existing in the database will not be able to ask the query.

The above explained foreign key constraints have been verified by executing the following queries in the workbench:

- INSERT INTO statements were used to insert queries in the *queries* table manually.
- **Query 1:**

```
INSERT INTO queries (person_role, person_id, message, email_id, query_date)
VALUES ("company_rep", 10098, "this is a yet another message",
        "prakkriti.saroj@iitgn.ac.in2", "15-04-2023");
```

- **Query 2:**

```
INSERT INTO queries (person_role, person_id, message, email_id, query_date)
VALUES ("company_rep", 98, "this is a yet another message",
        "prakkriti.saroj@iitgn.ac.in2", "15-04-2023");
```

- A person (company representative) with id 10098 exists in the database and thus gets added as shown in the figure below. However, there is no person with id 98 in the database and its insertion request violates the integrity constraint and hence is not added (shown as error in the figure below)

✓	14	18:24:30	use placement_management_system	0 row(s) affected	0.000 sec
✓	15	18:24:36	INSERT INTO queries (person_role, person_id, message, email_id, query_date) ...	2 row(s) affected Records: 2 Duplicates: 0 Warnings: 0	0.016 sec
✓	16	18:24:40	INSERT INTO queries (person_role, person_id, message, email_id, query_date) ...	1 row(s) affected	0.015 sec
✗	17	18:25:00	INSERT INTO queries (person_role, person_id, message, email_id, query_date) ...	Error Code: 1452. Cannot add or update a child row: a foreign key constraint fail...	0.016 sec

Fig.37

Functioning of the feature:

We have added a help option where students and company representatives can ask queries to the admin regarding the placement process. The page with this feature is shown below.

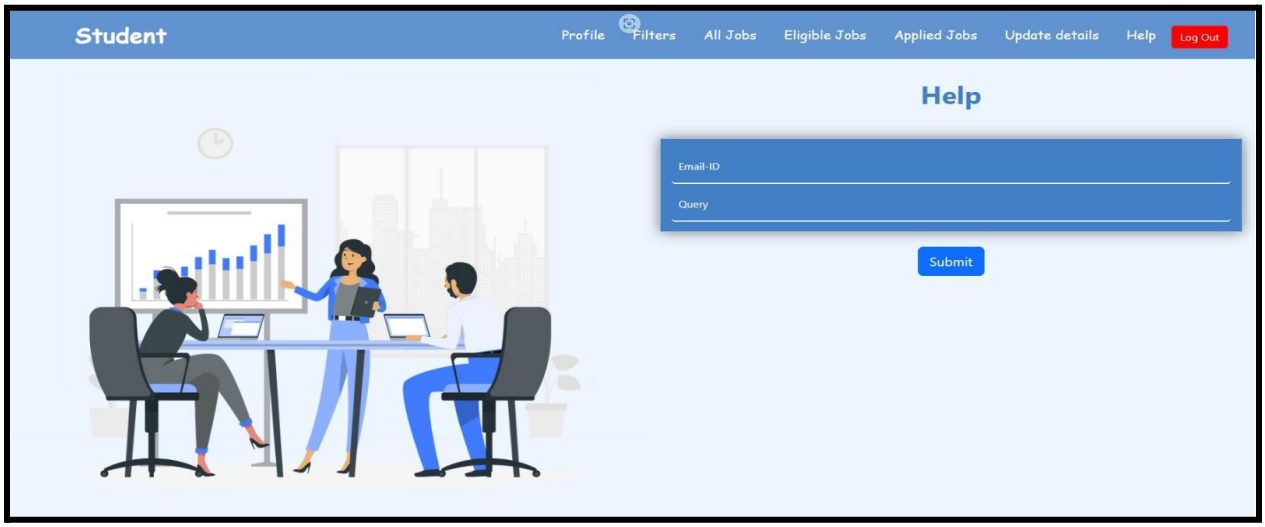


Fig.38

3.3 Responsibility of G1 & G2: 20 Pts

1. Documentation and screenshots of a total of 2 attacks [SQL Injection and XSS] performed and the defences against those attacks. (**15 points**)

Attack 1: SQL Injection:

Brief Overview:

SQL injection is a web security vulnerability where an agent can inject some malicious SQL statements via the web application which can in turn execute some SQL query that will result in displaying some sensitive information like fetching some table's data or dropping the table or database.

To prevent this attack, data entered by the user must be checked before executing some SQL query based on that.

We have identified a loophole for this attack in our web application. We have demonstrated the SQL injection attack and implemented its prevention in the following section.

Video of attack : link

The Person corresponding to person_id 10111 has name Oliver and has password Oliver111123. So for login we must put the correct person_id and password but if we put the person_id as 10111'-- we can login as Oliver irrespective of the correct password.

```
SELECT * FROM person WHERE person_id='10111'-- AND password_hash=' ';
```

The above query has to be executed when 10111'-- is given as input to person_id. This command in SQL considered everything after - - to be commented out. This means that if the User ID is 10111 then the user will be able to login no matter the password. This is also shown in the video.

Prevention:

Video of prevention of attack : link

Previously executed code:

```
        cur.execute(
            "SELECT * FROM person WHERE person_id=%s AND password_hash=%s", (user_id, password))
        user = cur.fetchone()
        if user:
            # successful login, redirect to home page
            session['loggedin'] = True
            session['id'] = user[0]
            return redirect(url_for('private'))
```

Code executed later:

```
cur.execute(
            "SELECT * FROM person WHERE person_id=%s AND password_hash=%s", (user_id, password))
        user = cur.fetchone()
        isAuthenticated = ("--" not in user_id and len(user_id)<10)
        if user and isAuthenticated:
            # successful login, redirect to home page
            session['loggedin'] = True
            session['id'] = user[0]
            session['ip_address'] = ip_address
            return redirect(url_for('private'))
```

We have fixed the length of person_id less than or equal to 10. This will ensure that the app will reject all the queries with length greater than 10. This will prevent the execution of any malicious queries because any such query will probably have a length greater than 10 as it has to be concatenated with a valid user_id. Moreover we have also made sure that there is no '--' in user_id as it is used for commenting the sql queries, so its presence might be vulnerable.

For example, an attacker can enter the query → "**10111; DROP TABLE address;**" Previously, we did not keep a check on the length of the entry of user_id. Hence, this query would have been executed and the address table could be dropped. Not only this, hackers could have entered any other query harmful to the database.

But now, we limit the length of 'user_id'. Hence, the above query will be rejected and the attack will fail.

Attack 2: XSS
Brief Overview:

An attacker can inject malicious code in a web page using XSS [Cross-Site Scripting]. They typically take advantage of the codebase or can persuade a person to fill a form or follow a link with hidden malicious code. Once they are successful in injecting the script they can steal sensitive information from the host's browser such as their login credentials or can impersonate the victim for performing illegal transactions.

Video of attack and its prevention : link

We are performing XSS by attribute injection. In our attack we exploited the unquoted attribute on our test page. While Jinja2 can protect you from XSS by escaping HTML, it cannot protect you from XSS by attribute injection.

Note : We have added an XSS button on the **admin profile page** for demonstrating the XSS attack since we did not have an XSS vulnerability in any of our pages!

Attack:
The code used before for the implementation purpose is written below and will trigger an alert when **tooltip onmouseover**=**alert**("**XSS**") is given as input in place of first name.

```
<h3> This will NOT trigger an alert </h3>
    <div style="align-items: normal;">
      <p value={{person[1]}}>{{person[1]}}<p>
      </div>
```

Prevention

For prevention, we can quote our attributes with either double or single quotes:

```
<h3> This will NOT trigger an alert </h3>
     <div style="align-items: normal;">
       <p value={{person[1]}}>{{person[1]}}<p>
       </div>
```

On hovering the mouse pointer over the name attribute below the "This will trigger an alert" text, the java script handler is executed.

On the other hand, on hovering the mouse pointer over the name attribute below the "This will NOT trigger an alert" text, the java script handler is not executed.

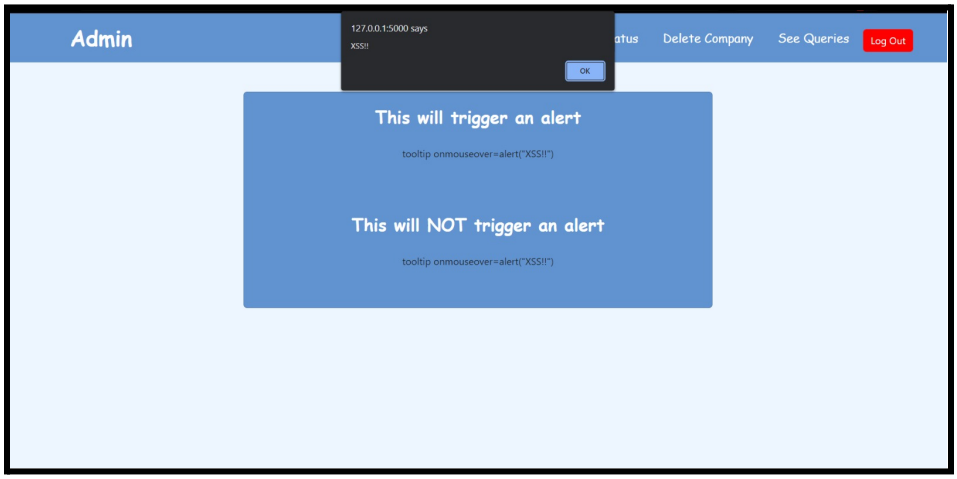


Fig.39

Alternative Approach: Disabling autoscaping in Jinja Template

An alternative way for injecting malicious Javascript code into our web app is by disabling auto escaping in our template. By default, Jinja renders special characters in the sense of HTML/XML like &,>,<," as well as " as text. On disabling autoescape we can directly inject HTML into pages.

Consider the following block of code

if the user enters a comment such as <**script**>**alert**("**XSS**")</**script**>, since we have disabled auto escaping it will run the javascript handler like before.

References:

- https://flask.palletsprojects.com/en/2.2.x/templating/
- https://flask.palletsprojects.com/en/2.2.x/security/

a. Additional attack and defence will lead to 10 **bonus** points for the team. *(Maximum 20 bonus points)*

Bonus attack 1: Unprotected Routes (G1)

Brief overview:

Unprotected routes are those that can be accessed by anyone without the need for user authentication or authorisation. These vulnerable sites may contain sensitive information about users/organisations such as private messages, user profiles, and so on. Attackers can obtain this information, resulting in serious security/privacy breaches. We discovered a vulnerability in our application and demonstrated the attack. We have also put in place preventative measures in response to this attack.

Attack:

Video of Attack: link

When a session is not implemented, the route becomes unprotected and thus, anyone can access the dashboard of a person by just entering the URL of the dashboard.

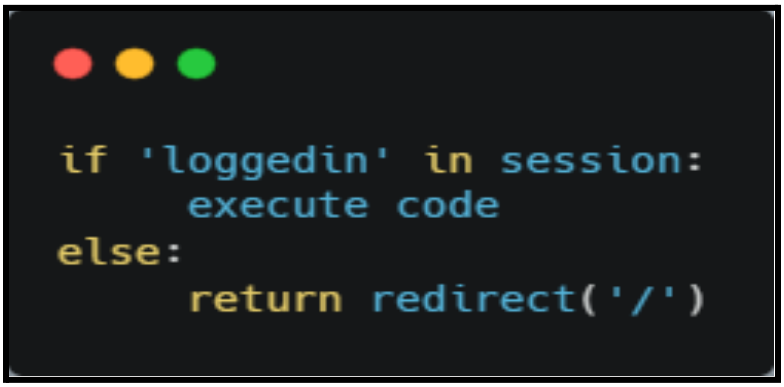
For example:

URL: http://localhost:5000/student-profile/10111 (10111 is the user_id of Oliver). Any person entering this query will be able to access the dashboard of Oliver and entering the URL: http://localhost:5000/student-profile/10116 (10116 is the user_id of Elijah) will be able to access the dashboard of Elijah.

Ideally, a person should access the dashboard only after login but with an unprotected route, anyone will be able to access the dashboard of a person by just knowing his/her user_id.

Prevention:

Video of prevention of attack : link



The above query is implemented to enable protected routes. A session is created when a person logs in and runs until he/she does not log out. Anyone trying to access the page using just the URL directly will be redirected to the login page and thus will not be able to access the dashboard of any other person. After the person logs out the session is terminated and thus no third party can access the dashboard without the required credentials (user_id and password).

Bonus attack 2: CSRF Attack (G2)

Brief overview:

Cross-Site Request Forgery is a web security vulnerability in which an attacker can perform actions impersonating another user without the user's consent. This attack can have severe repercussions like modification of user information (like

Email Ids), user data theft, actions performed on the behalf of the user. We have demonstrated this attack on our application and implemented its prevention.

Attack

Video of Attack: link

Previously we implemented the below code:

```
if 'loggedin' in session:
        cur = mysql.connection.cursor()
```

Here, if the user is logged in, the session starts and thus, he/she can access the dashboard. However, while the user is accessing the session here, he/she can change the id in the URL (For eg. http://localhost:5000/student-profile/10111 → http://localhost:5000/student-profile/10116 that is Oliver can change the user_id in URL to access the dashboard of Elijah). This can also be done using the inspect feature of the browser as shown in the video.

Prevention

Video of prevention of the attack: link

The code was changed as follows:

```
if 'loggedin' in session:
        user_id = session['id']
        if not has_access(user_id, person_id):
            error = "You are not allowed to access this page"
            return render_template('dashboard/std_error.html', error=error, person_id=user_id)
        cur = mysql.connection.cursor()
```

We implemented a function has_access shown below:

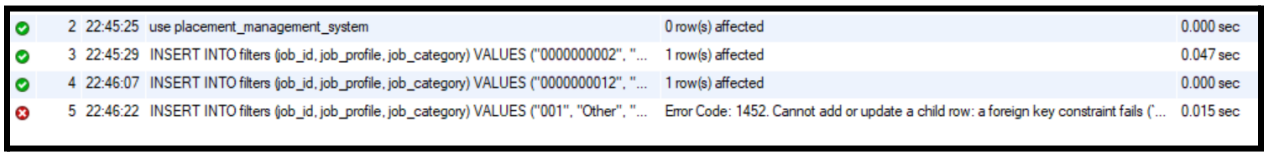
```
def has_access(user_id, person_id):
    user_id = str(user_id)
    print(user_id, person_id)
    for i in range(len(user_id)):
        if (user_id[i] != person_id[i]):
            return False
    return True
```

Here, has_access checks if the user_id of the person logged in is equal to the person_id of the page being accessed. If this is not the case, an error page will appear as shown in the video. Also, if a user tries to access the page using inspect, even then the error page will appear. Therefore, now the has_access function ensures that one particular user will only be able to access his/her own dashboard only and not any other's dashboard.

2. Show that all the relations and their constraints, finalised after the second feedback, are present and valid as per the ER diagram constructed in Assignment 1. (**5 points**)

Constraints:

- **Unique email** - Email ID has been kept unique for a particular person across all tables.
- **Unique id** - person_id/job_id will also be unique for a given person/job across all tables.
- **Integrity Errors** - As shown in the figure below, if a job_id does not exist in the job_profile table, it cannot be entered into the filters table. Here job_id 0000000002 exists in the job_profile table, hence it could be added in the filters table, but job_id 001 doesn't exist in the job_profile table, so it gave an error when we attempted to add it in the filters table. This is happening because job_profile is the parent table for filters, so if an ID is not present in job_profile, it cannot be added in filters.



✓	2	22:45:25	use placement_management_system	0 row(s) affected	0.000 sec
✓	3	22:45:29	INSERT INTO filters (job_id, job_profile, job_category) VALUES ("0000000002", "...	1 row(s) affected	0.047 sec
✓	4	22:46:07	INSERT INTO filters (job_id, job_profile, job_category) VALUES ("0000000012", "...	1 row(s) affected	0.000 sec
✗	5	22:46:22	INSERT INTO filters (job_id, job_profile, job_category) VALUES ("001", "Other", "...	Error Code: 1452. Cannot add or update a child row: a foreign key constraint fails (`...	0.015 sec

Fig.40

- **Update CPI:** Initially we have added a constraint in the database that the CPI must be greater than 0. Now if a student tries to update their CPI to zero, the CPI will not be updated.

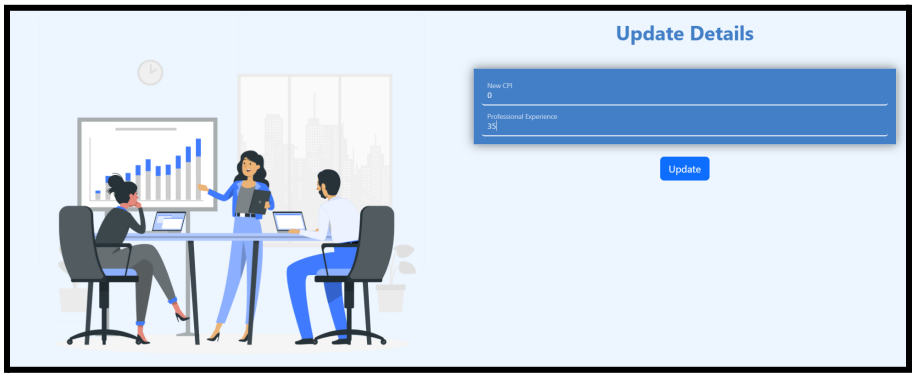


Fig.41

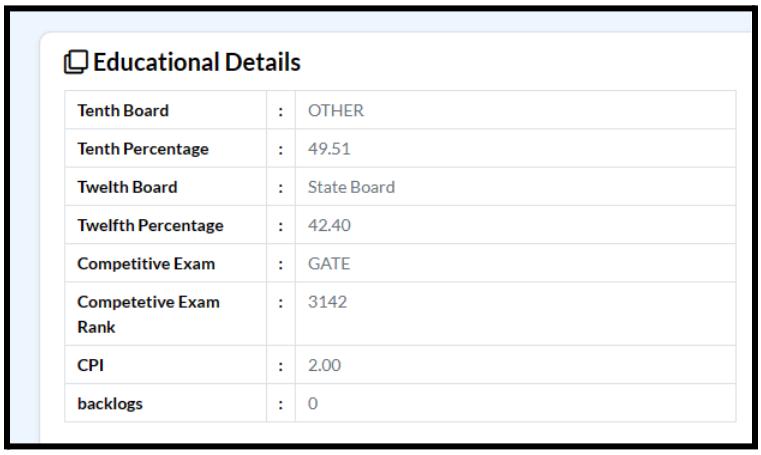


Fig.42

Like the above constraints we have maintained the other constraints mentioned in the previous assignments.

- **Modification in ER:** Two new **entities**, **queries** and **filters** and two new **relations** **person_queries** and **job_filters** :

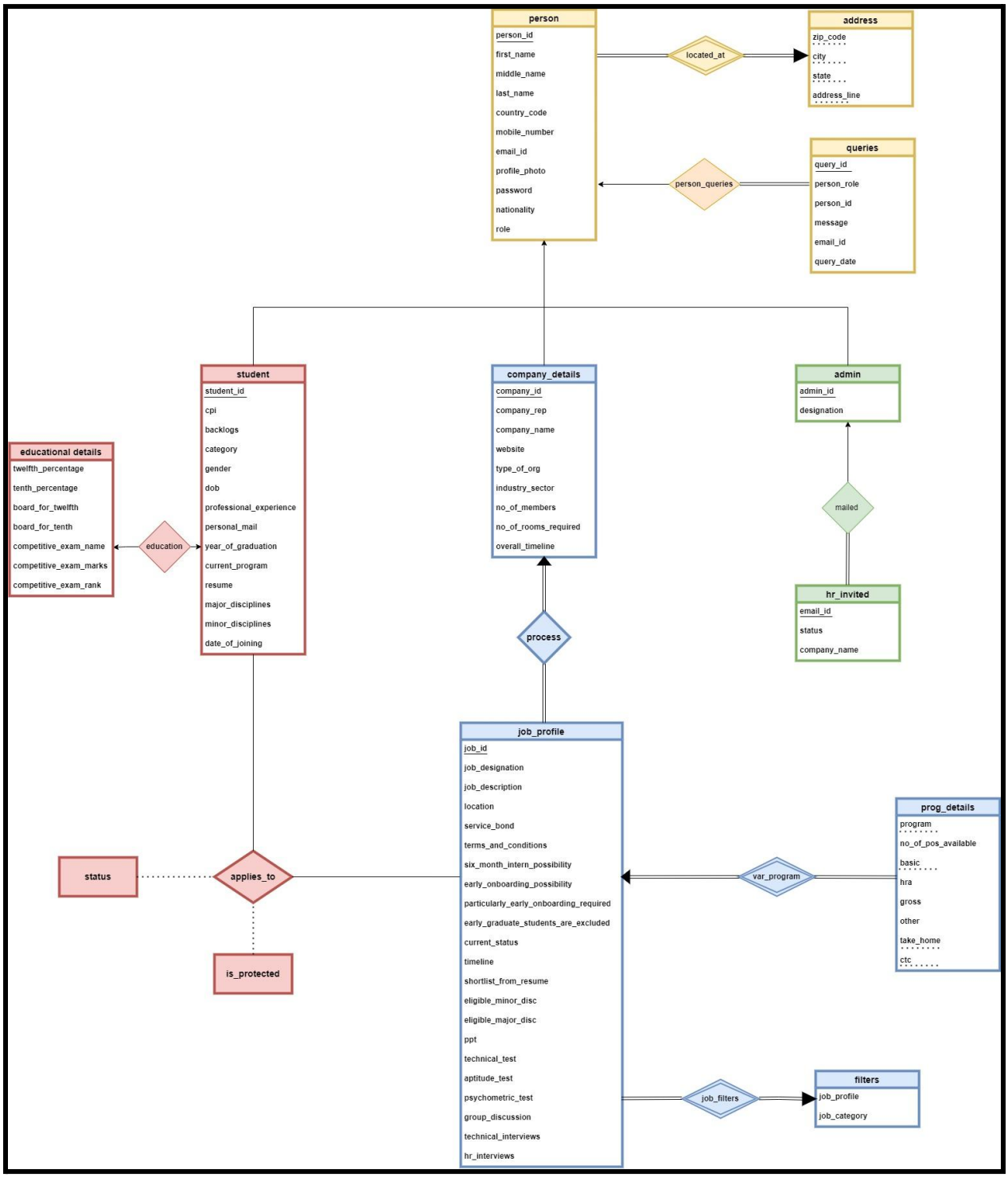


Fig.43

Members in G1:

1. Taha Syed
2. Aditi Dey
3. Hetvi Patel
4. Preetam Chimpa
5. Kevin Shah
6. Lipika Rajpal
7. Sagar Chavan
8. Rishab Jain
9. Samiksha Kamble

Members in G2:

1. Prakriti
2. Kamal Vaishnav

Individual Contributions:

Name	Contributions
Aditi Dey	G1; Helped in doing the SQL injection and XSS attacks. Made the UI for filters page. Contributed in writing the report for XSS attack and locks implementation
Hetvi Patel	G1; Assisted in taking feedback from various stakeholders. Contributed in making the final report and attacks.
Kamal Vaishnav	G2: integrated the UI with backend, helped in doing Unprotected routes and XSS attacks.
Kevin Shah	G1; Contacted various stakeholders and assisted in taking their feedback. Also assisted in making the final report and attacks.
Lipika Rajpal	G1; assisted in report writing, modifying the ER diagram and database schema, attacks
Prakriti	G2: integrated the UI with backend, helped in doing CSRF attack and SQL Injection attacks. Also implemented the locks.

	Updated the SQL files as required.
Preetam Chhimpa	G1; Making the Github readme file, making UI of admin related frontend, job-details, etc., report writing and formatting.
Rishab Jain	G1;Report writing and editing, made the dashboard for company, student as well as admin, made ui for login and resolved the issues in login, helped in making ui of all tables as well as the tables in the all tables page like job profile etc. Made responsive ui and implemented alerts and logout button.
Sagar Chavan	G1; Assisted in taking feedback from various stakeholders. Contributed in adding content to the report and formatting it.
Samiksha Kamble	G1, Helped in taking feedback from stakeholders. Assisted in writing the report. Made login page, registration forms, and profile views using HTML, CSS and JavaScript.
Taha Mohammad Syed	G1 contributed in front end of admin, company representative and student profile pages, demonstrating the XSS attack; documentation for the same.