Create a website according to the following requirements list:

1. Student and Company registration via email. Employees working for the same firm should be able to connect with each other (e.g. employees from PwC able to find each other's profile through pwc.de email) 2. Student profile builder and Company / Job profile builder, long with import function and location tracker 3. Options for companies to market themselves through feeds and posts. The followers' 'walls' should constitute of the contents from the companies they follow. 4. Based on "like", the students and the companies should be able to see the updates regarding the job. 5. Based on "like" the parties should be able to contact each other (exchange messages, documents, and make audio video calls). This feature need not be in-build in the first version. At the initial stage, external sites (e.g. Jitsi) can be used. 6. Matching based on "essential" and "desired" criteria in a scale (e.g. percentage match) highlighting the met and unmet criteria in the order of magnitude and excluding those not meeting the essential criteria. 7. Subscription offers and payment functions for the companies should be available. Admin should be able to make changes as per discretion. 8. Based on the subscription the companies should be able to upload quizzes, tests to screen out the candidates 9. CV downloader: The students should be able to convert their profile to a pdf file and download it. 10. Option of general and surgical advertisement placement e.g. a graduate trainee position for a Software Engineering should be visible to final year student of Software Engineering/Computer Science/IT students 11. Data sales: Data from 1-10 should be stored in a data bank and would be updated at regular intervals (e.g. monthly). The admin should be able grant access to users who will be able to download the data in their chosen format (e.g. xls). It is essential that the admin has the right to anonymise the data. 12. We would need 2 versions of the same product– 1 version for the students and another for the full-time job-seekers (this version would require some minor changes). 13. We would need the portals initially in English and German. Followed by French, Spanish, and other language. 14. The technology should be available in a form in which we can license it to partners in other countries. e.g. German version to Austria and Switzerland, and English version to the UK and Australia Licensing the technology.
ChatGPT

Creating a website that meets these comprehensive requirements involves several stages, from initial planning and design to development, testing, and deployment. Here's a high-level breakdown of the steps and key features for your website:

1. Registration and Networking

- **Email-based Registration**:
    - Implement email verification for registration.
    - Enable domain-based networking for company employees.

2. Profile Builder and Import Functionality

- **Student Profile Builder**:

    - Allow students to create detailed profiles, including education, skills, experiences, etc.
    - Include an import feature to pull information from LinkedIn or resumes.
    - Integrate a location tracker to show geographical information.
- **Company/Job Profile Builder**:
    - Enable companies to create profiles and post job listings.
    - Allow importing of company details from other platforms.

3. Marketing and Feed Features

- **Company Feeds and Posts**:
    - Allow companies to post updates, job openings, and other marketing materials.
    - Implement a follower system where students can follow companies.
    - Ensure student walls display posts from followed companies.

4. Interaction through Likes and Matches

- **Updates and Interaction**:
    - Show job updates and posts based on likes.
    - Enable message, document exchange, and AV calls through third-party services like Jitsi.

5. Matching System

- **Criteria-based Matching**:
    - Develop an algorithm to match students with jobs based on essential and desired criteria.
    - Highlight matches with a percentage score, showing unmet criteria.

6. Subscription and Payment Features

- **Subscription Models**:
    - Offer different subscription plans for companies.
    - Integrate a payment gateway for subscription transactions.
    - Allow admin to manage subscription plans and prices.

7. Screening Tests and Quizzes

- **Screening Functionality**:
    - Enable companies to upload and manage quizzes/tests for candidates.
    - Allow integration of test results with student profiles.

8. CV Downloader

- **PDF Generation**:
    - Implement functionality for students to download their profile as a formatted PDF CV.

9. Advertisement Placements

- **Targeted Advertisements**:
    - Allow companies to place general and targeted ads.
    - Ensure visibility to relevant student profiles (e.g., final-year students in specific fields).

10. Data Sales and Management

- **Data Bank**:
    - Store data securely with regular updates.
    - Provide admin-controlled access for data download in various formats (e.g., xls).
    - Include data anonymization features for privacy compliance.

11. Multiple Versions and Languages

- **Student vs. Job-Seeker Portals**:
    - Develop two versions of the portal with slight modifications for different user groups.
- **Multi-language Support**:
    - Start with English and German versions.
    - Plan for subsequent addition of French, Spanish, and other languages.

12. Licensing Technology

- **Licensing Framework**:
    - Develop a framework to license the technology for use in other countries.
    - Ensure easy customization for different regions (e.g., German version for Austria).

Technology Stack

- **Frontend**: React.js or Angular for a dynamic user interface.
- **Backend**: Node.js with Express or Django for robust backend support.
- **Database**: PostgreSQL or MongoDB for flexible and scalable data management.
- **Hosting**: AWS, Azure, or Google Cloud for reliable and scalable hosting solutions.
- **Payment Gateway**: Stripe or PayPal for handling transactions.
- **Messaging and Calls**: Integration with Jitsi or similar services.
- **Localization**: i18n for internationalization and localization.

Development Plan

1. **Requirement Analysis**: Detailed analysis of requirements and user stories.
2. **Design**: Create wireframes and prototypes for the website.
3. **Development**:
    - Set up the development environment.
    - Develop registration and profile management features.
    - Implement company feeds and post features.
    - Develop matching and interaction functionalities.
    - Integrate payment gateways and subscription models.
    - Implement data management and export features.
    - Create CV download functionality.
4. **Testing**: Rigorous testing (unit, integration, and user acceptance testing).
5. **Deployment**: Deploy on a scalable cloud platform.
6. **Maintenance**: Ongoing support and feature updates.

This comprehensive plan covers the main aspects required to build the website, ensuring it meets all specified requirements and provides a robust, user-friendly experience for both students and companies.

4o
Generate a proof of concept in python.
7/8
ChatGPT

Creating a proof of concept (PoC) for a complex website like the one described involves demonstrating key functionalities in a simplified form. For this PoC, we'll focus on:

1. **User Registration**: Basic email-based registration for students and companies.
2. **Profile Creation**: Simple profile builder for students and companies.
3. **Job Posting**: Ability for companies to post jobs.
4. **Basic Matching**: Simple matching mechanism based on essential and desired criteria.
5. **Data Export**: Export user profiles and job posts to CSV for demonstration.

We'll use Python with Flask for the web framework, SQLite for the database, and basic HTML for the frontend. This PoC will be kept minimal to illustrate core functionalities.

Prerequisites

Install Flask: Install Flask and required libraries.
bash
Copy code

```
pip install Flask Flask-SQLAlchemy
```

1.

Create Project Structure:
plaintext
Copy code

```
/poc
  |-- app.py
  |-- models.py
  |-- templates/
      |-- register.html
      |-- profile.html
      |-- post_job.html
      |-- match.html
      |-- export.html
```

2.

Step-by-Step Implementation

1. `app.py` - Main Application File

python
Copy code

```
from flask import Flask, request, redirect, url_for, render_template,
flash
from flask_sqlalchemy import SQLAlchemy
import pandas as pd

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///poc.db'
app.config['SECRET_KEY'] = 'secret_key_here'
db = SQLAlchemy(app)

from models import Student, Company, JobPost

# Home Route
@app.route('/')
def home():
    return render_template('register.html')

# Register Student
@app.route('/register_student', methods=['POST'])
def register_student():
    email = request.form['email']
    name = request.form['name']
    if Student.query.filter_by(email=email).first():
        flash('Student already registered.')
    else:
        student = Student(email=email, name=name)
        db.session.add(student)
        db.session.commit()
        flash('Student registered successfully.')
    return redirect(url_for('home'))

# Register Company
@app.route('/register_company', methods=['POST'])
def register_company():
    email = request.form['email']
    name = request.form['name']
    if Company.query.filter_by(email=email).first():
        flash('Company already registered.')
    else:
        company = Company(email=email, name=name)
        db.session.add(company)
        db.session.commit()
        flash('Company registered successfully.')
    return redirect(url_for('home'))

# Post Job
@app.route('/post_job', methods=['GET', 'POST'])
def post_job():
    if request.method == 'POST':
        title = request.form['title']
        description = request.form['description']
        company_id = request.form['company_id']
        job = JobPost(title=title, description=description,
company_id=company_id)
        db.session.add(job)
        db.session.commit()
        flash('Job posted successfully.')
        return redirect(url_for('post_job'))
    companies = Company.query.all()
    return render_template('post_job.html', companies=companies)

# Match
@app.route('/match', methods=['GET'])
def match():
    # Dummy matching for demonstration
    students = Student.query.all()
    jobs = JobPost.query.all()
    matches = []
    for student in students:
        for job in jobs:
            # Simple matching logic based on dummy criteria
            if 'Python' in job.description:
                matches.append((student.name, job.title))
    return render_template('match.html', matches=matches)

# Export Data
@app.route('/export', methods=['GET'])
def export_data():
    students = Student.query.all()
    jobs = JobPost.query.all()
    student_data = [(s.name, s.email) for s in students]
    job_data = [(j.title, j.description) for j in jobs]
    df_students = pd.DataFrame(student_data, columns=['Name',
'Email'])
    df_jobs = pd.DataFrame(job_data, columns=['Title', 'Description'])
    df_students.to_csv('students.csv', index=False)
    df_jobs.to_csv('jobs.csv', index=False)
    return render_template('export.html')

if __name__ == '__main__':
    db.create_all()
    app.run(debug=True)
```

2. `models.py` - Database Models

python
Copy code

```
from app import db

class Student(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(120), unique=True, nullable=False)
    name = db.Column(db.String(80), nullable=False)

class Company(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(120), unique=True, nullable=False)
    name = db.Column(db.String(80), nullable=False)

class JobPost(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(120), nullable=False)
    description = db.Column(db.Text, nullable=False)
    company_id = db.Column(db.Integer, db.ForeignKey('company.id'),
nullable=False)
    company = db.relationship('Company', backref=db.backref('jobs',
lazy=True))
```

3. HTML Templates

register.html:

html
Copy code

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Register</title>
</head>
<body>
    <h1>Register</h1>
    <form action="{{ url_for('register_student') }}" method="post">
        <h2>Register Student</h2>
        <input type="text" name="name" placeholder="Name" required>
        <input type="email" name="email" placeholder="Email" required>
        <button type="submit">Register</button>
    </form>
    <form action="{{ url_for('register_company') }}" method="post">
        <h2>Register Company</h2>
        <input type="text" name="name" placeholder="Name" required>
        <input type="email" name="email" placeholder="Email" required>
        <button type="submit">Register</button>
    </form>
</body>
</html>
```

post_job.html:

html
Copy code

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Post Job</title>
</head>
<body>
    <h1>Post a Job</h1>
    <form action="{{ url_for('post_job') }}" method="post">
        <input type="text" name="title" placeholder="Job Title"
required>
        <textarea name="description" placeholder="Job Description"
required></textarea>
        <select name="company_id" required>
            {% for company in companies %}
                <option value="{{ company.id }}">{{ company.name
}}</option>
            {% endfor %}
        </select>
        <button type="submit">Post Job</button>
    </form>
</body>
</html>
```

match.html:

html
Copy code
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Match</title>
</head>
<body>
    <h1>Match Results</h1>
    <ul>
        {% for student, job in matches %}
            <li>{{ student }} matched with job {{ job }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

export.html:

html
Copy code
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Export Data</title>
</head>
<body>
    <h1>Data Exported</h1>
    <p>Students and Jobs data have been exported to CSV files.</p>
</body>
</html>
```

Running the PoC

Initialize the Database: Run `app.py` once to create the database schema.
bash
Copy code

```
python app.py
```

1.
2. **Access the Application**: Open your browser and go to `http://127.0.0.1:5000`.

This PoC showcases a simplified version of the required features. For a production-grade application, you'd need to implement additional features such as robust security measures, user authentication, more complex matching algorithms, and support for multiple languages and localization.

4o mini
AttributeError: module 'sqlalchemy' has no attribute '__all__'.
ChatGPT

The `AttributeError` you're encountering typically suggests that there's a problem with the SQLAlchemy import or setup in your Flask application. This error might occur due to incorrect usage of SQLAlchemy or version incompatibilities.

Let's troubleshoot and fix this issue step-by-step:

1. Verify SQLAlchemy Installation

Ensure you have the correct version of SQLAlchemy installed. Run:

bash
Copy code

```
pip show SQLAlchemy
```

Check if SQLAlchemy is installed and note its version. If it's missing or an outdated version, update or install it using:

bash
Copy code

```
pip install --upgrade SQLAlchemy
```

2. Correct Import Statements

Ensure you are using correct import statements. In your `app.py` file, you should import `SQLAlchemy` from `flask_sqlalchemy` and use it properly. Here's the updated code:

`app.py`:

python

Copy code

```
from flask import Flask, request, redirect, url_for, render_template,
flash
from flask_sqlalchemy import SQLAlchemy
import pandas as pd

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///poc.db'
app.config['SECRET_KEY'] = 'secret_key_here'
db = SQLAlchemy(app)

from models import Student, Company, JobPost

@app.route('/')
def home():
    return render_template('register.html')

@app.route('/register_student', methods=['POST'])
def register_student():
    email = request.form['email']
    name = request.form['name']
    if Student.query.filter_by(email=email).first():
        flash('Student already registered.')
    else:
        student = Student(email=email, name=name)
        db.session.add(student)
        db.session.commit()
        flash('Student registered successfully.')
    return redirect(url_for('home'))

@app.route('/register_company', methods=['POST'])
def register_company():
    email = request.form['email']
    name = request.form['name']
    if Company.query.filter_by(email=email).first():
        flash('Company already registered.')
    else:
        company = Company(email=email, name=name)
        db.session.add(company)
        db.session.commit()
        flash('Company registered successfully.')
    return redirect(url_for('home'))

@app.route('/post_job', methods=['GET', 'POST'])
def post_job():
    if request.method == 'POST':
        title = request.form['title']
        description = request.form['description']
        company_id = request.form['company_id']
        job = JobPost(title=title, description=description,
company_id=company_id)
        db.session.add(job)
        db.session.commit()
        flash('Job posted successfully.')
        return redirect(url_for('post_job'))
    companies = Company.query.all()
    return render_template('post_job.html', companies=companies)

@app.route('/match', methods=['GET'])
def match():
    students = Student.query.all()
    jobs = JobPost.query.all()
    matches = []
    for student in students:
        for job in jobs:
            if 'Python' in job.description:
                matches.append((student.name, job.title))
    return render_template('match.html', matches=matches)

@app.route('/export', methods=['GET'])
def export_data():
    students = Student.query.all()
    jobs = JobPost.query.all()
    student_data = [(s.name, s.email) for s in students]
    job_data = [(j.title, j.description) for j in jobs]
    df_students = pd.DataFrame(student_data, columns=['Name',
'Email'])
    df_jobs = pd.DataFrame(job_data, columns=['Title', 'Description'])
    df_students.to_csv('students.csv', index=False)
    df_jobs.to_csv('jobs.csv', index=False)
    return render_template('export.html')

if __name__ == '__main__':
    db.create_all()
    app.run(debug=True)
```

3. Check Your `models.py` File

Ensure your `models.py` file correctly imports `db` from the `app.py` module.

`models.py`:

python
Copy code

```
from app import db

class Student(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(120), unique=True, nullable=False)
    name = db.Column(db.String(80), nullable=False)

class Company(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.String(120), unique=True, nullable=False)
    name = db.Column(db.String(80), nullable=False)

class JobPost(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(120), nullable=False)
    description = db.Column(db.Text, nullable=False)
    company_id = db.Column(db.Integer, db.ForeignKey('company.id'),
nullable=False)
    company = db.relationship('Company', backref=db.backref('jobs',
lazy=True))
```

4. Check for Circular Imports

Make sure there are no circular imports between `app.py` and `models.py`. Import `db` from `app` in `models.py` but avoid importing models into `app.py`.

5. Run the Application

Once you’ve ensured the above steps are correct, run the application:

bash
Copy code
```
python app.py
```

This should resolve the `AttributeError`. If issues persist, ensure that your Python environment is clean and reinstall packages if necessary. If you encounter any other specific errors, feel free to share them for further assistance.

4o mini
Gerate an app.config['SECRET_KEY'] = 'secret_key_here'
ChatGPT

In a Flask application, the `SECRET_KEY` is used for various cryptographic operations, including session management and CSRF protection. It's important to generate a strong, random secret key for security purposes. Here's how you can generate a `SECRET_KEY`:

Generating a Secure Secret Key

You can generate a strong secret key using Python's `secrets` module, which provides functions for generating cryptographically secure random numbers. Here's a simple way to generate a secure key:

Using Python's `secrets` Module

Open a Python interpreter or create a script file:
python
Copy code
```
import secrets
print(secrets.token_hex(16))
```

1. This will generate a secure, random hexadecimal string of 32 characters (16 bytes).

Run the script:
bash
Copy code

```
python generate_secret_key.py
```

If you run this code, you'll get an output like:
Copy code

```
5d41402abc4b2a76b9719d911017c592
```

2.

Use the generated key in your Flask app:
Copy the generated key and set it in your Flask configuration. For example:
python
Copy code

```
app.config['SECRET_KEY'] = '5d41402abc4b2a76b9719d911017c592'
```

3.

Example in Flask App

Here's how you would integrate the generated `SECRET_KEY` into your `app.py`:

python
Copy code

```
from flask import Flask, request, redirect, url_for, render_template, 
flash
from flask_sqlalchemy import SQLAlchemy
import pandas as pd
import secrets

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///poc.db'

# Generate a secure secret key
app.config['SECRET_KEY'] = secrets.token_hex(16)

db = SQLAlchemy(app)

from models import Student, Company, JobPost

@app.route('/')
def home():
    return render_template('register.html')

@app.route('/register_student', methods=['POST'])
def register_student():
    email = request.form['email']
    name = request.form['name']
    if Student.query.filter_by(email=email).first():
        flash('Student already registered.')
    else:
        student = Student(email=email, name=name)
        db.session.add(student)
        db.session.commit()
        flash('Student registered successfully.')
    return redirect(url_for('home'))

@app.route('/register_company', methods=['POST'])
def register_company():
    email = request.form['email']
    name = request.form['name']
    if Company.query.filter_by(email=email).first():
        flash('Company already registered.')
    else:
        company = Company(email=email, name=name)
        db.session.add(company)
        db.session.commit()
        flash('Company registered successfully.')
    return redirect(url_for('home'))

@app.route('/post_job', methods=['GET', 'POST'])
def post_job():
    if request.method == 'POST':
        title = request.form['title']
        description = request.form['description']
        company_id = request.form['company_id']
        job = JobPost(title=title, description=description,
company_id=company_id)
        db.session.add(job)
        db.session.commit()
        flash('Job posted successfully.')
        return redirect(url_for('post_job'))
    companies = Company.query.all()
    return render_template('post_job.html', companies=companies)

@app.route('/match', methods=['GET'])
def match():
    students = Student.query.all()
    jobs = JobPost.query.all()
    matches = []
    for student in students:
        for job in jobs:
            if 'Python' in job.description:
                matches.append((student.name, job.title))
    return render_template('match.html', matches=matches)

@app.route('/export', methods=['GET'])
def export_data():
    students = Student.query.all()
    jobs = JobPost.query.all()
    student_data = [(s.name, s.email) for s in students]
    job_data = [(j.title, j.description) for j in jobs]
    df_students = pd.DataFrame(student_data, columns=['Name',
'Email'])
    df_jobs = pd.DataFrame(job_data, columns=['Title', 'Description'])
    df_students.to_csv('students.csv', index=False)
    df_jobs.to_csv('jobs.csv', index=False)
    return render_template('export.html')

if __name__ == '__main__':
    db.create_all()
    app.run(debug=True)
```

Additional Considerations

1. **Environment Variables**: For production environments, it's good practice to store the `SECRET_KEY` in environment variables rather than hard-coding it in your application. This can be done using tools like `python-dotenv` to manage environment variables.

2. **Security**: Make sure the `SECRET_KEY` is kept private and not exposed in your version control system or logs. It should be unique for each application and kept secure.

4o mini
generate web pages in python to query the data.
2/2
ChatGPT

To create web pages for querying data using Python and Flask, you'll need to build pages that allow users to input queries and view the results. For this example, let's create a basic web application where users can:

1. Query students by email.
2. Query job posts by title.
3. View results on separate pages.

Updated Project Structure

We'll add additional HTML templates and update the Flask application code to handle these queries.

Project Structure:

plaintext
Copy code
```
/poc
  |-- app.py
  |-- models.py
  |-- templates/
      |-- register.html
      |-- profile.html
      |-- post_job.html
      |-- match.html
      |-- export.html
      |-- query_students.html
      |-- query_jobs.html
      |-- query_results.html
```

Updated `app.py`

Here is how you can update your `app.py` to include routes for querying and displaying data:

python
Copy code

```
from flask import Flask, request, redirect, url_for, render_template,
flash
from flask_sqlalchemy import SQLAlchemy
import pandas as pd
import secrets

app = Flask(__name__)
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///poc.db'
app.config['SECRET_KEY'] = secrets.token_hex(16)
db = SQLAlchemy(app)

from models import Student, Company, JobPost

@app.route('/')
def home():
    return render_template('register.html')

@app.route('/register_student', methods=['POST'])
def register_student():
    email = request.form['email']
    name = request.form['name']
    if Student.query.filter_by(email=email).first():
        flash('Student already registered.')
    else:
        student = Student(email=email, name=name)
        db.session.add(student)
        db.session.commit()
        flash('Student registered successfully.')
    return redirect(url_for('home'))

@app.route('/register_company', methods=['POST'])
def register_company():
    email = request.form['email']
    name = request.form['name']
    if Company.query.filter_by(email=email).first():
        flash('Company already registered.')
    else:
        company = Company(email=email, name=name)
        db.session.add(company)
        db.session.commit()
        flash('Company registered successfully.')
    return redirect(url_for('home'))

@app.route('/post_job', methods=['GET', 'POST'])
def post_job():
    if request.method == 'POST':
        title = request.form['title']
        description = request.form['description']
        company_id = request.form['company_id']
        job = JobPost(title=title, description=description,
company_id=company_id)
        db.session.add(job)
        db.session.commit()
        flash('Job posted successfully.')
        return redirect(url_for('post_job'))
    companies = Company.query.all()
    return render_template('post_job.html', companies=companies)

@app.route('/match', methods=['GET'])
def match():
    students = Student.query.all()
    jobs = JobPost.query.all()
    matches = []
    for student in students:
        for job in jobs:
            if 'Python' in job.description:
                matches.append((student.name, job.title))
    return render_template('match.html', matches=matches)

@app.route('/export', methods=['GET'])
def export_data():
    students = Student.query.all()
    jobs = JobPost.query.all()
    student_data = [(s.name, s.email) for s in students]
    job_data = [(j.title, j.description) for j in jobs]
    df_students = pd.DataFrame(student_data, columns=['Name',
'Email'])
    df_jobs = pd.DataFrame(job_data, columns=['Title', 'Description'])
    df_students.to_csv('students.csv', index=False)
    df_jobs.to_csv('jobs.csv', index=False)
    return render_template('export.html')

# Query Students
@app.route('/query_students', methods=['GET', 'POST'])
def query_students():
    if request.method == 'POST':
        email = request.form['email']
        student = Student.query.filter_by(email=email).first()
        return render_template('query_results.html', results=[student]
if student else [], query='Student')
    return render_template('query_students.html')

# Query Jobs
@app.route('/query_jobs', methods=['GET', 'POST'])
def query_jobs():
    if request.method == 'POST':
        title = request.form['title']
        jobs =
JobPost.query.filter(JobPost.title.like(f'%{title}%')).all()
        return render_template('query_results.html', results=jobs,
query='Job')
    return render_template('query_jobs.html')

if __name__ == '__main__':
    db.create_all()
    app.run(debug=True)
```

HTML Templates

`query_students.html`

A form to query students by email:

html

Copy code

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Query Students</title>
</head>
<body>
    <h1>Query Students</h1>
    <form action="{{ url_for('query_students') }}" method="post">
        <label for="email">Enter Student Email:</label>
        <input type="email" name="email" id="email" required>
        <button type="submit">Search</button>
    </form>
    <a href="{{ url_for('home') }}">Home</a>
    <a href="{{ url_for('query_jobs') }}">Query Jobs</a>
</body>
</html>
```

`query_jobs.html`

A form to query jobs by title:

html
Copy code

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Query Jobs</title>
</head>
<body>
    <h1>Query Jobs</h1>
    <form action="{{ url_for('query_jobs') }}" method="post">
        <label for="title">Enter Job Title:</label>
        <input type="text" name="title" id="title" required>
        <button type="submit">Search</button>
    </form>
    <a href="{{ url_for('home') }}">Home</a>
    <a href="{{ url_for('query_students') }}">Query Students</a>
</body>
</html>
```

query_results.html

Display the query results:

html
Copy code

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Query Results</title>
</head>
<body>
    <h1>Query Results for {{ query }}</h1>
    {% if results %}
        <ul>
            {% for result in results %}
                <li>
                    {% if query == 'Student' %}
                        Name: {{ result.name }}, Email: {{
result.email }}
                    {% elif query == 'Job' %}
                        Title: {{ result.title }}, Description: {{
result.description }}
                    {% endif %}
                </li>
            {% endfor %}
        </ul>
    {% else %}
        <p>No results found.</p>
    {% endif %}
    <a href="{{ url_for('query_students') }}">Query Students</a>
    <a href="{{ url_for('query_jobs') }}">Query Jobs</a>
    <a href="{{ url_for('home') }}">Home</a>
</body>
```

```
</html>
```

Summary

In this updated version of the Flask application:

- **`query_students.html`**: Provides a form to search for students by email.
- **`query_jobs.html`**: Provides a form to search for jobs by title.
- **`query_results.html`**: Displays the results of the queries, showing either student or job information based on the query type.

This setup allows you to query and view data from the database in a simple and effective manner using Flask and HTML templates.

4o mini
create buttons on main webpage allowing to query the data.
ChatGPT

To add buttons on the main webpage allowing users to navigate to the data query pages, you will need to update your `register.html` template to include these buttons.

Here's how you can modify `register.html` to include buttons for querying students and jobs:

Updated `register.html`

html
Copy code

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Main Page</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            text-align: center;
            margin: 50px;
        }
        .button {
            display: inline-block;
            padding: 10px 20px;
            margin: 10px;
            font-size: 16px;
            text-decoration: none;
            color: white;
            background-color: #007bff;
            border-radius: 5px;
            border: none;
            cursor: pointer;
        }
        .button:hover {
            background-color: #0056b3;
        }
        h1 {
            margin-bottom: 30px;
        }
    </style>
</head>
<body>
    <h1>Welcome to the Data Management System</h1>

    <a href="{{ url_for('query_students') }}" class="button">Query
Students</a>
    <a href="{{ url_for('query_jobs') }}" class="button">Query
Jobs</a>

    <!-- Add other navigation links as needed -->
    <br>
    <a href="{{ url_for('post_job') }}" class="button">Post Job</a>
    <a href="{{ url_for('match') }}" class="button">Match</a>
    <a href="{{ url_for('export_data') }}" class="button">Export
Data</a>

</body>
</html>
```

Explanation:

1. **Styling**: Added basic CSS to style the buttons and ensure they look clean and are easy to click.
    - `.button`: Defines the style for the buttons, including padding, font size, background color, border radius, and cursor style.
    - `.button:hover`: Changes the button color when hovered over.
2. **Navigation Links**: Added buttons linking to different routes:
    - `Query Students`: Links to the `/query_students` route.
    - `Query Jobs`: Links to the `/query_jobs` route.
    - Additional buttons are included for other functionalities such as posting jobs, viewing matches, and exporting data.
3. **HTML Structure**: Used semantic HTML elements to structure the content and buttons on the page.

Updating Flask Routes (for Reference)

Make sure your Flask app (`app.py`) has the routes defined for querying students and jobs:

python
Copy code

```
@app.route('/query_students', methods=['GET', 'POST'])
def query_students():
    if request.method == 'POST':
        email = request.form['email']
        student = Student.query.filter_by(email=email).first()
        return render_template('query_results.html', results=[student]
if student else [], query='Student')
    return render_template('query_students.html')

@app.route('/query_jobs', methods=['GET', 'POST'])
def query_jobs():
    if request.method == 'POST':
        title = request.form['title']
        jobs =
JobPost.query.filter(JobPost.title.like(f'%{title}%')).all()
        return render_template('query_results.html', results=jobs,
query='Job')
    return render_template('query_jobs.html')
```

Summary

With these changes, your main page (`register.html`) will now include buttons that allow users to easily navigate to pages where they can query students and jobs. This improves user experience by providing clear navigation options and a modern, clean design.