

AI-Powered Job Interview Simulation System

Step 1: Start with the basic Flask app for questions and answers

1.1 Create the basic folder structure as follows:

```
ai_job_interview/  
├── app.py  
├── templates/  
│   └── index.html  
├── static/  
│   └── style.css
```

1.2 Write the basic backend logic (Flask) in app.py:

```
from flask import Flask, render_template, request, jsonify  
  
app = Flask(__name__)  
  
# Predefined interview questions  
questions = [  
    "Tell me about yourself.",  
    "What are your strengths?",  
    "Why should we hire you?",  
    "Describe a challenge you faced and how you overcame it."  
]  
  
# Route to display the first question  
@app.route('/')  
def index():  
    return render_template('index.html', question=questions[0])  
  
# Route to handle user's answer and give feedback  
@app.route('/submit_answer', methods=['POST'])  
def submit_answer():  
    answer = request.json['answer']  
    feedback = "Good answer!" if len(answer) > 0 else "Please provide an answer."  
    return jsonify({"feedback": feedback})  
  
if __name__ == '__main__':  
    app.run(debug=True)
```

This basic Flask app will:

- Show a question on the webpage (index.html).
- Allow the user to submit an answer, and show feedback based on the answer's length.

1.3 Create a simple **HTML page** in `templates/index.html` to display questions and get answers:

```
<!DOCTYPE html>
<html>
<head>
  <title>Job Interview Simulation</title>
  <link rel="stylesheet" href="/static/style.css">
</head>
<body>
  <div class="container">
    <h1>Job Interview</h1>
    <p id="question">{{ question }}</p>
    <textarea id="answer" placeholder="Your answer here..."></textarea>
    <button onclick="submitAnswer()">Submit Answer</button>
    <div id="feedback"></div>
  </div>

  <script>
    function submitAnswer() {
      const answer = document.getElementById('answer').value;
      fetch('/submit_answer', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json'
        },
        body: JSON.stringify({ answer: answer })
      })
      .then(response => response.json())
      .then(data => {
        document.getElementById('feedback').innerText = data.feedback;
      });
    }
  </script>
</body>
</html>
```

This HTML page shows the first interview question and allows the user to type an answer in a text box.

- After submitting the answer, a simple **feedback message** is displayed.

Step 2: Running the basic app

1. Open your terminal and **navigate to your project folder** (`ai_job_interview`).
2. Run the Flask app:

python app.py

3. Go to <http://127.0.0.1:5000> in your browser.

You should see:

- An interview question on the page.
- A text area to type an answer.
- When you submit the answer, you should get feedback below the answer box.

NOW, we can start adding the next features one by one:

1. Add Voice Input (Speech-to-Text)

We will integrate **voice input** using the **Web Speech API**. This allows the user to **speak their answer** instead of typing.

1.1 Update `index.html`:

Add a button to start voice input and integrate the Web Speech API to capture speech.

```
<!DOCTYPE html>
<html>
<head>
  <title>Job Interview Simulation</title>
  <link rel="stylesheet" href="/static/style.css">
</head>
<body>
  <div class="container">
    <h1>Job Interview</h1>
    <p id="question">{{ question }}</p>
    <textarea id="answer" placeholder="Your answer here..."></textarea>
    <button onclick="submitAnswer()">Submit Answer</button>
    <button onclick="startVoiceInput()">Start Voice Input</button>
    <div id="feedback"></div>
  </div>

  <script>
    let recognition;

    // Start voice input using the Web Speech API
    function startVoiceInput() {
      if (!('webkitSpeechRecognition' in window)) {
        alert("Your browser does not support voice input.");
        return;
      }

      recognition = new webkitSpeechRecognition();
      recognition.continuous = false;
      recognition.interimResults = false;
```

```

    recognition.onstart = function() {
        console.log("Voice input started...");
    };

    recognition.onresult = function(event) {
        let voiceInput = event.results[0][0].transcript;
        document.getElementById('answer').value = voiceInput; // Fill text box with voice input
    };

    recognition.onerror = function(event) {
        console.log("Error occurred in speech recognition: " + event.error);
    };

    recognition.start();
}

// Submit the answer (either typed or via voice)
function submitAnswer() {
    const answer = document.getElementById('answer').value;
    fetch('/submit_answer', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({ answer: answer })
    })
    .then(response => response.json())
    .then(data => {
        document.getElementById('feedback').innerText = data.feedback;
    });
}
</script>
</body>
</html>

```

Explanation:

- **Voice input button:** The "Start Voice Input" button starts the **speech recognition** when clicked.
- When the user speaks, the Web Speech API listens and converts the voice into text, which is then filled into the answer text box.

2. Save Answers & Scores to a File (CSV)

Now, let's store the answers and feedback in a **CSV file** for later use.

2.1 Modify app.py to save answers to a CSV:

```
import csv
```

```
# Function to save answers and scores to CSV
def save_to_csv(answer, score, feedback):
```

```

with open('interview_results.csv', mode='a', newline='') as file:
    writer = csv.writer(file)
    writer.writerow([answer, score, feedback])

# Route to handle user's answer and give feedback
@app.route('/submit_answer', methods=['POST'])
def submit_answer():
    data = request.json
    answer = data['answer']
    feedback = "Good answer!" if len(answer) > 0 else "Please provide an answer."

    # Save the answer, feedback, and score to CSV
    save_to_csv(answer, len(answer), feedback)

    return jsonify({"feedback": feedback})

```

Explanation:

- **save_to_csv function:** This function appends the user's answer, score (based on length), and feedback to a **CSV file** called `interview_results.csv`.

3. Add User Login

Let's add a basic **user authentication system** using **Flask-Login** so that users can log in before starting the interview.

3.1 Install Flask-Login:

First, install the **Flask-Login** library:

pip install flask-login

3.2 Modify app.py to include login functionality:

```

from flask_login import LoginManager, UserMixin, login_user,
login_required, logout_user, current_user
from flask import Flask, render_template, request, redirect,
url_for

app = Flask(__name__)

# Set up Flask-Login
login_manager = LoginManager()
login_manager.init_app(app)

class User(UserMixin):
    def __init__(self, id):
        self.id = id

@login_manager.user_loader
def load_user(user_id):
    return User(user_id)

# Simple route for login

```

```

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        user_id = request.form['username']
        user = User(user_id)
        login_user(user)
        return redirect('/')
    return render_template('login.html')

# Protect interview page with login
@app.route('/')
@login_required
def index():
    return render_template('index.html', question=questions[0])

# Route to handle logout
@app.route('/logout')
@login_required
def logout():
    logout_user()
    return redirect('/login')

```

3.3 Create **login.html**:

This is a simple login form where the user can enter their **username**.

```

<!DOCTYPE html>

<html>

<head>

    <title>Login</title>

</head>

<body>

    <h2>Login</h2>

    <form method="POST">

        <label for="username">Username:</label>

        <input type="text" id="username" name="username" required>

        <button type="submit">Login</button>

    </form>

</body>

</html>

```

Explanation:

- **Login Route:** Users will be able to log in before starting the interview. After login, they will be redirected to the interview page.

- **Protected Interview Route:** The interview page is now **protected**, and users must be logged in to access it.

4. Track Progress or Performance Over Time

We will use **Chart.js** to track the interview performance by visualizing scores over time.

4.1 Add Chart.js to index.html:

```
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
<canvas id="progressChart" width="400" height="200"></canvas>
```

4.2 Update JavaScript to track scores:

```
let scores = [];

function submitAnswer() {
  const answer = document.getElementById('answer').value;
  fetch('/submit_answer', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json'
    },
    body: JSON.stringify({ answer: answer })
  })
  .then(response => response.json())
  .then(data => {
    document.getElementById('feedback').innerText = data.feedback;

    // Track scores
    scores.push(data.feedback === "Good answer!" ? 10 : 0);
    updateChart();
  });
}
```

```
function updateChart() {
  var ctx = document.getElementById('progressChart').getContext('2d');
  var progressChart = new Chart(ctx, {
```

```

type: 'line',
data: {
  labels: Array.from({ length: scores.length }, (_, i) => `Q${i + 1}`),
  datasets: [{
    label: 'Score Progress',
    data: scores,
    borderColor: 'rgba(75, 192, 192, 1)',
    borderWidth: 2
  }]
}
});
}

```

Explanation:

- **scores array**: Stores the scores for each question (10 for good answers, 0 for others).
- **Chart.js**: Visualizes the progress of the interview by showing how the user's score changes with each question.

5. Resume Analyzer (Bonus AI Feature)

We will add a simple feature where users can upload their **resume** (PDF) and extract **skills** from it.

5.1 Install PyPDF2:

```
pip install PyPDF2
```

5.2 Modify `app.py` to include resume analysis:

```

import PyPDF2

@app.route('/analyze_resume', methods=['POST'])
def analyze_resume():
    resume = request.files['file']
    pdf_reader = PyPDF2.PdfReader(resume)
    text = ""
    for page in pdf_reader.pages:
        text += page.extract_text()

```



```
skills = ['Python', 'Java', 'SQL', 'JavaScript']  
found_skills = [skill for skill in skills if skill.lower() in text.lower()]
```

```
return jsonify({'skills_found': found_skills})
```

5.3 Update `index.html` to allow file uploads:

```
<input type="file" id="resume-upload" accept=".pdf" />  
<button onclick="analyzeResume()">Analyze Resume</button>
```

5.4 Add JavaScript to handle file upload:

```
function analyzeResume() {  
    var file = document.getElementById('resume-upload').files[0];  
    var formData = new FormData();  
    formData.append('file', file);  
  
    fetch('/analyze_resume', {  
        method: 'POST',  
        body: formData  
    })  
    .then(res => res.json())  
    .then(data => {  
        alert('Skills found: ' + data.skills_found.join(', '));  
    });  
}
```

Explanation:

- **File input:** The user can upload their resume as a PDF.
- **Resume analysis:** We extract text from the PDF and check if any **predefined skills** are mentioned (e.g., Python, JavaScript).

Full Working Resume Analysis with Skill Matching

Assuming you have a route for resume upload (say `/upload_resume`), here's the complete continuation:

In `app.py`:

```
from flask import Flask, render_template, request, redirect, url_for
```

```

from PyPDF2 import PdfReader
import csv
import os

app = Flask(__name__)

@app.route('/upload_resume', methods=['POST'])
def upload_resume():
    if 'resume' not in request.files:
        return "No file uploaded.", 400

    resume_file = request.files['resume']
    if resume_file.filename == "":
        return "No selected file.", 400

    # Read text from the uploaded PDF
    pdf_reader = PdfReader(resume_file)
    resume_text = ""
    for page in pdf_reader.pages:
        resume_text += page.extract_text()

    # Skill Matching Logic
    skills = ['Python', 'JavaScript', 'HTML', 'CSS', 'Machine Learning']
    found_skills = [skill for skill in skills if skill.lower() in resume_text.lower()]
    score = len(found_skills) / len(skills) * 100
    feedback = f"Skills matched: {' '.join(found_skills)}"

    # Save the result (optional)
    with open("resume_analysis.csv", "a", newline="") as file:
        writer = csv.writer(file)
        writer.writerow([resume_file.filename, ' '.join(found_skills), f"{score:.2f}%"])

```

```
# Render result page with analysis

return render_template("resume_result.html", score=score, feedback=feedback,
found_skills=found_skills)
```

```
# Main route just for testing upload form
```

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

```
def home():
```

```
    return render_template("upload_resume.html")
```

templates/upload_resume.html

```
<!DOCTYPE html>

<html>

<head>

    <title>Upload Resume</title>

</head>

<body>

    <h2>Upload Resume PDF</h2>

    <form action="/upload_resume" method="post" enctype="multipart/form-data">

        <input type="file" name="resume" accept=".pdf" required>

        <button type="submit">Upload</button>

    </form>

</body>

</html>
```

templates/resume_result.html

```
<!DOCTYPE html>

<html>

<head>

    <title>Resume Analysis Result</title>

</head>

<body>

    <h2>Resume Analysis</h2>

    <p><strong>Score:</strong> {{ score }}%</p>
```

```

<p><strong>Matched Skills:</strong> {{ feedback }}</p>
{% if found_skills|length == 0 %}
    <p>No relevant skills found. Try adding technical skills like Python, HTML, etc.</p>
{% endif %}
<a href="/">Upload Another Resume</a>
</body>
</html>

```

Now when you upload a resume PDF:

- It will extract text
- Match skills
- Show score + feedback
- Save data in CSV

Install Required Packages

Run this command in terminal to install required dependencies:

pip install flask PyPDF2

Then create a `requirements.txt`:

flask

PyPDF2

Run the App

Run your project:

python app.py

Visit: <http://127.0.0.1:5000>

Once this is working, the next step is to connect this to the interview simulation questions.

Step 6: Start Interview Simulation After Resume Analysis

After we analyze the resume, we'll **automatically redirect the user** to an interview simulation page where they'll be asked AI-generated or predefined questions based on their skills.

✅ Step-by-Step Plan:

1. Create a new route `/start_interview`
2. Create a new template `start_interview.html`

3. Automatically redirect user to this page after resume analysis
4. Show skill-based interview questions
5. (Optional) Collect answers and provide feedback

Update app.py — Add Interview Simulation Route

Add this route at the bottom of app.py

```
@app.route('/start_interview')
def start_interview():
    # Sample skill-based questions (based on resume score or found_skills)
    questions = [
        "Tell me about a project you built using Python.",
        "What is the difference between HTML and CSS?",
        "Can you explain how Machine Learning works?",
        "What are the key concepts of JavaScript?",
        "How do you optimize a website's performance?"
    ]
    return render_template("start_interview.html", questions=questions)
```

Modify Redirect in analyze_resume Route

Right now the code ends like this:

```
return render_template("resume_result.html", ...)
```

Change it to redirect to interview:

```
return redirect(url_for("start_interview"))
```

Now after resume analysis, it will go to /start_interview.

Create templates/start_interview.html

```
<!DOCTYPE html>
<html>
<head><title>Interview Questions</title></head>
<body>
    <h2>Interview Simulation</h2>
    <form action="/submit_answers" method="post">
```

```

{% for question in questions %}

<div>

    <p><strong>Q{{ loop.index }}:</strong> {{ question }}</p>

    <textarea name="answer{{ loop.index }}" rows="4" cols="50" required></textarea>

</div>

{% endfor %}

<br>

<button type="submit">Submit Answers</button>

</form>

</body>

</html>

```

+ Add Answer Submission Route

Also in app.py:

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

```
def submit_answers():
```

```
    answers = []
```

```
    for i in range(1, 6): # Assuming 5 questions
```

```
        answer = request.form.get(f'answer{i}')
```

```
        answers.append(answer)
```

```
# Here you can evaluate answers using AI later or just display them
```

```
return render_template("interview_feedback.html", answers=answers)
```

Create templates/interview_feedback.html

```

<!DOCTYPE html>

<html>

<head><title>Interview Feedback</title></head>

<body>

    <h2>Your Answers</h2>

    {% for answer in answers %}

    <div>

        <p><strong>Answer {{ loop.index }}:</strong> {{ answer }}</p>

```

```

    </div>

{% endfor %}






<p>Thank you for participating!</p>

</body>

</html>

```

Now you have a complete flow:

1. User uploads resume 
2. Skills are analyzed 
3. User is redirected to interview questions 
4. User submits answers 
5. Feedback page shown 

Step 7: Add AI Feedback to Interview Answers (using OpenAI or Local Logic)

This step will simulate an interviewer giving **intelligent feedback** based on the user's answers.

 **Option 1: Use GPT for Real Feedback (requires OpenAI key)**

 **Option 2: Use Keyword Matching for Local Feedback (simple logic)**

We'll start with **Option 2: Local Keyword-Based Feedback** since it doesn't require an API key.

Step-by-Step for Local Feedback System:

1. Update `submit_answers` route in `app.py`:

Replace your `submit_answers` code with this:

```

@app.route('/submit_answers', methods=['POST'])
def submit_answers():
    feedbacks = []
    questions = [
        "Tell me about a project you built using Python.",
        "What is the difference between HTML and CSS?",
        "Can you explain how Machine Learning works?",
        "What are the key concepts of JavaScript?",
        "How do you optimize a website's performance?"
    ]
    keywords = [
        ['project', 'python', 'developed', 'code'],
        ['html', 'css', 'style', 'structure'],
        ['data', 'algorithm', 'model', 'training'],
        ['variables', 'functions', 'DOM', 'events'],
        ['optimize', 'load', 'speed', 'cache']
    ]

```

```
]
```

```
for i in range(5):
    answer = request.form.get(f'answer{i+1}').lower()
    key_hits = sum(1 for word in keywords[i] if word in answer)
    if key_hits >= 3:
        fb = "Good answer! You covered the key points."
    elif key_hits == 2:
        fb = "Fair answer. You touched on some important topics."
    else:
        fb = "Needs improvement. Try to be more specific."
    feedbacks.append((questions[i], answer, fb))

return render_template("interview_feedback.html", feedbacks=feedbacks)
```

2. Update `interview_feedback.html`

```
<!DOCTYPE html>
<html>
<head><title>Interview Feedback</title></head>
<body>
    <h2>Your Interview Feedback</h2>
    {% for q, a, f in feedbacks %}
        <div>
            <p><strong>Q{{ loop.index }}: {{ q }}</strong></p>
            <p><strong>Your Answer:</strong> {{ a }}</p>
            <p><strong>Feedback:</strong> {{ f }}</p>
            <hr>
        </div>
    {% endfor %}
    <a href="/">Back to Home</a>
</body>
</html>
```

Test It:

- Try vague answers → you'll get “Needs improvement.”
- Use technical keywords → you'll get “Good answer!”

Step 8: Add User Authentication (Login & Signup)

This allows users to:

- Sign up with email & password
- Log in and access their interview dashboard
- Secure access to resume analysis & interview
- Track their performance later (future upgrade)

⚙️ Tech Stack Used:

- **Flask** (Python backend)

- **Flask-Login** or session-based auth
- **HTML/CSS/JS** for frontend

Step-by-Step Implementation:

1. Install Required Package

Install `Flask` if you haven't:

`pip install flask`

2. Update `app.py` – Add User Authentication

Add these imports at the top:

`from flask import Flask, render_template, request, redirect, url_for, session`

Then, initialize a secret key:

`app.secret_key = 'your_secret_key' # Replace with something secure`

Add a dummy user store (in-memory for now):

`users = {'test@example.com': '123456'} # email: password`

3. Add Signup & Login Routes

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

`def signup():`

```
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']
        if email in users:
            return "User already exists!"
        users[email] = password
        return redirect(url_for('login'))
    return render_template('signup.html')
```

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

`def login():`

```
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']
        if users.get(email) == password:
            session['user'] = email
            return redirect(url_for('home'))
        else:
            return "Invalid credentials"
    return render_template('login.html')
```

`@app.route('/logout')`

`def logout():`

```
    session.pop('user', None)
    return redirect(url_for('login'))
```

4. Create `signup.html` and `login.html`

templates/signup.html

```
<h2>Signup</h2>
<form method="POST">
  Email: <input type="email" name="email" required><br>
  Password: <input type="password" name="password" required><br>
  <button type="submit">Signup</button>
</form>
```

templates/login.html

```
<h2>Login</h2>
<form method="POST">
  Email: <input type="email" name="email" required><br>
  Password: <input type="password" name="password" required><br>
  <button type="submit">Login</button>
</form>
```

5. Restrict Access to Pages (like resume upload)

Update any route like this:

@app.route('/home')

def home():

if 'user' not in session:

return redirect(url_for('login'))





return render_template('index.html')

Do this check for `/upload`, `/start_interview`, etc.

6. Add Logout Button in Navbar or Home


`Logout`




Now users can:

- Sign up 
- Log in 
- Only access tools after login 
- Log out 

Step 9: Save User Data & Interview Results in a Database

To make the system smarter and persistent, we'll **store**:

-  User details (email, password)

-  Resume score & skills
-  Interview answers
-  Feedback & scores

Step 1: Install `mysql-connector-python`

pip install mysql-connector-python

Step 2: Create a MySQL Database

1. Log in to MySQL:

mysql -u root -p

2. Run these commands:

```
CREATE DATABASE interview_system;  
USE interview_system;
```


```
CREATE TABLE users (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  email VARCHAR(255) UNIQUE NOT NULL,  
  password VARCHAR(255) NOT NULL  
);
```

```
CREATE TABLE results (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  user_email VARCHAR(255),  
  resume_score INT,  
  skills_found TEXT,  
  answers TEXT,  
  feedback TEXT  
);
```

Step 3: Connect Flask to MySQL

In your `app.py`, add this:

```
import mysql.connector  
  
def get_db_connection():  
  return mysql.connector.connect(  
    host='localhost',  
    user='your_mysql_username',  
    password='your_mysql_password',  
    database='interview_system'  
  )
```

 Replace `your_mysql_username` and `your_mysql_password` with your credentials.

Step 4: Update Signup Code

```
@app.route('/signup', methods=['GET', 'POST'])
def signup():
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']

        conn = get_db_connection()
        cursor = conn.cursor()
        try:
            cursor.execute("INSERT INTO users (email, password) VALUES (%s, %s)", (email,
password))
            conn.commit()
        except mysql.connector.IntegrityError:
            return "User already exists!"
        conn.close()
        return redirect(url_for('login'))
    return render_template('signup.html')
```

Step 5: Update Login Code

```
@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']

        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT * FROM users WHERE email = %s AND password = %s",
(email, password))
        user = cursor.fetchone()
        conn.close()

        if user:
            session['user'] = email
            return redirect(url_for('home'))
        else:
            return "Invalid credentials"
    return render_template('login.html')
```

Step 6: Update Result Saving

In your `/submit_answers` route:

```
conn = get_db_connection()
cursor = conn.cursor()
cursor.execute(
    "INSERT INTO results (user_email, resume_score, skills_found, answers, feedback)
VALUES (%s, %s, %s, %s, %s)",
    (email, 0, '', '|'.join(all_answers), '|'.join(all_feedback))
)
```

```
conn.commit()
conn.close()
```

Step 10: Show Previous Interview Results (MySQL Version)

What We'll Do:

1. Add a route `/results` to fetch interview history
2. Query the `results` table for the logged-in user
3. Display the data nicely in an HTML table

1. Add Route in `app.py`

```
@app.route('/results')
```

```
def results():
```

```
    if 'user' not in session:
```

```
        return redirect(url_for('login'))
```

```
    email = session['user']
```

```
    conn = get_db_connection()
```

```
    cursor = conn.cursor()
```

```
    cursor.execute("SELECT resume_score, skills_found, answers, feedback FROM results WHERE  
user_email = %s", (email,))
```

```
    rows = cursor.fetchall()
```

```
    conn.close()
```

```
    # Prepare clean data
```

```
    data = []
```

```
    for row in rows:
```

```
        resume_score = row[0]
```

```
        skills_found = row[1]
```

```
        answers = row[2].split('|')
```

```
        feedback = row[3].split('|')
```

```
        data.append({
```

```
            'resume_score': resume_score,
```

```
            'skills_found': skills_found,
```

```
        'answers': answers,
        'feedback': feedback
    })
```

```
return render_template('results.html', data=data)
```

2. Create **results.html** in Templates

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Your Interview Results</title>

    <style>

        body { font-family: Arial, sans-serif; padding: 20px; }

        table { width: 100%; border-collapse: collapse; margin-bottom: 40px; }

        th, td { padding: 10px; border: 1px solid #ccc; }

        h2 { color: #2c3e50; }

    </style>

</head>

<body>

    <h1>Your Previous Interview Attempts</h1>

    {% for attempt in data %}

        <h2>Attempt:</h2>

        <p><strong>Resume Score:</strong> {{ attempt.resume_score }}</p>

        <p><strong>Skills Found:</strong> {{ attempt.skills_found }}</p>

        <table>

            <tr>

                <th>Answer</th>

                <th>Feedback</th>

            </tr>

            {% for ans, fb in zip(attempt.answers, attempt.feedback) %}
```

```

        <tr>

            <td>{{ ans }}</td>

            <td>{{ fb }}</td>

        </tr>

    {% endfor %}

</table>

{% endfor %}

</body>

</html>

```

3. Add a Link to Results Page

In your homepage (`home.html` or `dashboard.html`), add:

```
<a href="/results"> 📁 View My Interview History</a>
```

Step 11: PDF Report Generation

We'll use a Python library called **report lab** or **xhtml2pdf** to generate styled PDFs from user data.

Here we'll go with **xhtml2pdf** (easier for HTML → PDF).

1. Install **xhtml2pdf**

pip install xhtml2pdf

2. Create a Route to Generate the PDF

In `app.py`:

```
from xhtml2pdf import pisa
```

```
from io import BytesIO
```

```
from flask import make_response
```

```
@app.route('/download_report')
```

```
def download_report():
```

```
    if 'user' not in session:
```

```
        return redirect(url_for('login'))
```

```

email = session['user']
conn = get_db_connection()
cursor = conn.cursor()

cursor.execute("SELECT resume_score, skills_found, answers, feedback FROM results
WHERE user_email = %s ORDER BY id DESC LIMIT 1", (email,))

result = cursor.fetchone()
conn.close()

```

if not result:

```

    return "No interview result found."

```

```

resume_score, skills_found, answers, feedback = result

```

```

answers = answers.split('|')

```

```

feedback = feedback.split('|')

```

Prepare HTML content for PDF

```

html = render_template("pdf_template.html", resume_score=resume_score,
skills_found=skills_found,
                        answers=answers, feedback=feedback)

```

```

pdf = BytesIO()

```

```

pisa_status = pisa.CreatePDF(html, dest=pdf)

```

if pisa_status.err:

```

    return "PDF generation failed"

```

```

pdf.seek(0)

```

```

return send_file(pdf, download_name="interview_report.pdf", as_attachment=True)

```

3. Create pdf_template.html in /templates

```

<!DOCTYPE html>

```

```

<html>

```

```

<head>

```



```

<meta charset="UTF-8">
<title>Interview Report</title>
<style>
    body { font-family: Arial, sans-serif; }
    h1, h2 { color: #333; }
    table { width: 100%; border-collapse: collapse; margin-top: 20px; }
    th, td { padding: 8px; border: 1px solid #999; text-align: left; }
</style>
</head>
<body>
    <h1>Interview Summary Report</h1>
    <p><strong>Resume Score:</strong> {{ resume_score }}%</p>
    <p><strong>Skills Found:</strong> {{ skills_found }}</p>

    <h2>Interview Q&A + Feedback</h2>
    <table>
        <tr>
            <th>Answer</th>
            <th>Feedback</th>
        </tr>
        {% for ans, fb in zip(answers, feedback) %}
        <tr>
            <td>{{ ans }}</td>
            <td>{{ fb }}</td>
        </tr>
        {% endfor %}
    </table>
</body>
</html>

```

4. Add Link to Download PDF

In your `results.html` or `home.html`:

`↓ Download My Interview Report (PDF)`

Users can now download their own personalized interview reports as PDFs!

Step 12: Deployment (with **Render** – easiest free option)

We'll deploy the full Flask + MySQL system online

Option: Render (Free Hosting for Flask + MySQL)

1. Prepare Your Project for Deployment

Your directory should look like this:

```
/project-root
├── app.py
├── templates/
├── static/
├── requirements.txt
├── database.sql (optional)
└── config.py
```

2. Create `requirements.txt`

In terminal:

```
pip freeze > requirements.txt
```

Make sure it includes:

```
Flask
mysql-connector-python
xhtml2pdf
```

You can clean up unnecessary lines manually if needed.

3. Add a `config.py` File

```
# config.py
DB_HOST = "your-mysql-host"
DB_USER = "your-user"
DB_PASSWORD = "your-password"
DB_NAME = "your-db-name"
SECRET_KEY = "any-secret-key"
```

Update `app.py` to import config:

```
import config
app.secret_key = config.SECRET_KEY
```

Update MySQL connection:

```
def get_db_connection():
    return mysql.connector.connect(
        host=config.DB_HOST,
        user=config.DB_USER,
        password=config.DB_PASSWORD,
        database=config.DB_NAME
    )
```

4. Push Project to GitHub

1. Create a GitHub repo
2. Upload your project files
3. Make sure `.gitignore` includes:

```
__pycache__/  
*.pyc  
.env
```

5. Deploy on Render

1. Go to <https://render.com/>
2. Click “New Web Service” → Connect your GitHub
3. Fill in:

- **Name:** ai-job-interview-system
- **Runtime:** Python
- **Start command:**

gunicorn app:app

Environment: Python 3.x

- **Build command** (optional): `pip install -r requirements.txt`

4. Add environment variables:

- DB_HOST
- DB_USER
- DB_PASSWORD
- DB_NAME
- SECRET_KEY

5. Deploy 🚀

6. Done!

You'll get a live URL like:

<https://ai-job-interview-system.onrender.com>

Step 13: Create a Visual Dashboard with Chart.js

1. Install Chart.js

No installation needed — just include it in your HTML via CDN:

In your `dashboard.html` (new file in `/templates`):

```
<script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
```

2. Create `dashboard.html` Template

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Interview Dashboard</title>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <style>
    body { font-family: Arial; padding: 20px; background: #f7f7f7; }
    canvas { max-width: 600px; margin: 40px auto; display: block; }
    h2 { text-align: center; }
  </style>
</head>
<body>
  <h2> 📊 Your Interview Dashboard</h2>

  <canvas id="resumeChart"></canvas>
  <canvas id="skillsChart"></canvas>
  <canvas id="feedbackChart"></canvas>

  <script>
    const resumeData = {
      labels: ['Resume Score', 'Remaining'],
      datasets: [{
        label: 'Resume Score',
        data: [{ resume_score }, { 100 - resume_score }],
        backgroundColor: ['#4caf50', '#ccc']
      }]
    };
    new Chart(document.getElementById('resumeChart'), {
      type: 'doughnut',
      data: resumeData,
    });

    const skills = {{ skills_list|tojson }};
    const skillsChart = {
```

```

        labels: skills,
        datasets: [{
            label: 'Matched Skills',
            data: skills.map(() => 1),
            backgroundColor: '#2196f3'
        }]
    };
    new Chart(document.getElementById('skillsChart'), {
        type: 'bar',
        data: skillsChart,
    });

    const feedbackData = {{ feedback_scores|tojson }};
    new Chart(document.getElementById('feedbackChart'), {
        type: 'line',
        data: {
            labels: feedbackData.map((_, i) => 'Q' + (i + 1)),
            datasets: [{
                label: 'Feedback Score',
                data: feedbackData,
                fill: false,
                borderColor: '#f44336'
            }]
        }
    });
</script>
</body>
</html>

```

3. Add Route in app.py

```

@app.route('/dashboard')
def dashboard():
    if 'user' not in session:
        return redirect(url_for('login'))

    email = session['user']
    conn = get_db_connection()
    cursor = conn.cursor()
    cursor.execute("SELECT resume_score, skills_found, feedback FROM results WHERE
user_email = %s ORDER BY id DESC LIMIT 1", (email,))
    result = cursor.fetchone()
    conn.close()

    if not result:
        return "No results found."

    resume_score, skills_found, feedback = result
    skills_list = skills_found.split(',')
    feedback_scores = []

    for item in feedback.split('|'):

```

```

try:
    # Simple logic: longer feedback = higher score
    score = min(len(item) / 20, 10)
    feedback_scores.append(round(score, 2))
except:
    feedback_scores.append(5)

return render_template('dashboard.html', resume_score=resume_score,
                       skills_list=skills_list, feedback_scores=feedback_scores)

```

4. Add Dashboard Link in Navbar / Home Page

`  View My Dashboard`

Now users can visually understand:

- How strong their resume is
- Which skills matched
- How well they answered each question

Step 14: Secure User Login System with Hashed Passwords

1. Install Werkzeug (if not already included)

pip install werkzeug

2. Update `register.html` (if you haven't created it yet)

```

<!-- templates/register.html -->
<form method="POST" action="/register">
    <input type="text" name="email" placeholder="Email" required />
    <input type="password" name="password" placeholder="Password" required />
    <button type="submit">Register</button>
</form>

```

3. Add Registration Logic with Hashing in `app.py`

```
from werkzeug.security import generate_password_hash, check_password_hash
```

```

@app.route('/register', methods=['GET', 'POST'])
def register():
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']

        hashed_password = generate_password_hash(password)

        conn = get_db_connection()
        cursor = conn.cursor()

```

```

        cursor.execute("SELECT * FROM users WHERE email = %s", (email,))
        existing = cursor.fetchone()
        if existing:
            return "User already exists."

        cursor.execute("INSERT INTO users (email, password) VALUES (%s, %s)", (email,
hashed_password))
        conn.commit()
        conn.close()

        return redirect(url_for('login'))

    return render_template('register.html')

```

4. Update Login Logic to Use check_password_hash

```

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        email = request.form['email']
        password = request.form['password']

        conn = get_db_connection()
        cursor = conn.cursor()
        cursor.execute("SELECT password FROM users WHERE email = %s", (email,))
        user = cursor.fetchone()
        conn.close()

        if user and check_password_hash(user[0], password):
            session['user'] = email
            return redirect(url_for('home'))
        else:
            return "Invalid credentials."

    return render_template('login.html')

```

5. Update Your MySQL users Table

```

CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(255) UNIQUE,
    password TEXT
);

```

6. (Optional) Logout Route

```

@app.route('/logout')
def logout():
    session.pop('user', None)
    return redirect(url_for('login'))

```

You now have a Secure Auth System!

- Passwords are **hashed**, not stored in plain text
- Session is used to manage login/logout
- Works well with your existing dashboard & system