INDEED JOB SCRAPER

1.Abstract

This project presents the design and development of a **job portal web application** that aggregates job postings and allows users to search for relevant opportunities. The portal is implemented using the Flask web framework in Python, with data stored in a JSON file. The jobs data was collected through web scraping techniques and later integrated into the system.

The application provides users with the ability to filter jobs by **title** and **location**, thereby narrowing down the search space to find positions that match their interests. The frontend interface is built using **HTML and CSS**, styled for responsiveness and clarity. Unlike static listings, the application dynamically updates search results based on user queries, making it highly interactive.

This system demonstrates how simple but powerful technologies such as Flask, JSON, and modern UI design can be combined to create a fully functional prototype of a job search portal.

2. Introduction

The rapid growth of online job opportunities has resulted in scattered listings across multiple platforms, making it challenging for job seekers to identify suitable openings efficiently. A dedicated job portal acts as a bridge between job seekers and recruiters by consolidating listings into a single accessible interface.

This project was developed to showcase a **minimal yet effective job portal** using the Flask framework in Python. The application reads job postings from a JSON dataset generated via scraping. Users can search jobs using keywords (for example, *Python Developer*, *Backend Engineer*) and locations (for example, *Remote*, *Bangalore*). The results are displayed in a grid-based card layout, which includes job title, company name, and location.

The aim of this project is not only to demonstrate the power of Flask in building lightweight web applications but also to highlight the importance of structuring data and presenting it effectively to the end user.

3. System Requirements

Hardware Requirements

- Minimum 4 GB RAM (8 GB recommended for running scraper tools smoothly)
- Intel i3 Processor or equivalent (i5 or above recommended)
- 500 MB storage for project files and dependencies

Software Requirements

- **Operating System**: Windows 10/Linux/Mac
- **Programming Language**: Python 3.8 or higher
- **Framework**: Flask (micro web framework for Python)
- **Libraries**: JSON (for handling job data), Jinja2 (template rendering)
- **Tools**: Any IDE (VS Code, PyCharm, or Sublime Text), Browser (Chrome/Firefox)
- **Package Manager**: pip for installing Flask and dependencies

This minimal requirement ensures that the project can run on almost any system with modern specifications.

4. Project Architecture

The project follows a **client–server architecture** using Flask as the backend server.

- **Data Layer**: Jobs are stored in a jobs.json file, which acts as a local database.
- **Application Layer (Flask)**: Flask handles user requests, loads data, filters results, and renders responses.
- **Presentation Layer (HTML + CSS)**: Job listings and search results are displayed in an interactive interface styled with CSS grid layouts.

Flow of Execution:

1. User accesses the portal via the homepage (/).
2. A search query or location is entered in the search form.
3. Flask loads jobs.json and filters records based on query/location.
4. Filtered job listings are passed to the template engine (Jinja2).
5. The HTML page renders results in job cards format.

This modular design separates logic (backend) from presentation (frontend), ensuring scalability and maintainability.

5. Implementation

job-portal/

|— app.py

|— jobs.json

|— templates/

| └— jobs-1.html

|— static/

| ├— style.css

| └— logo.png

Flask Backend (app.py)

The backend is responsible for:

- Loading data from the JSON file.
- Handling search queries from the user.
- Filtering results by keyword and location.
- Rendering the results page with filtered jobs.

Frontend (HTML + CSS)

The frontend uses:

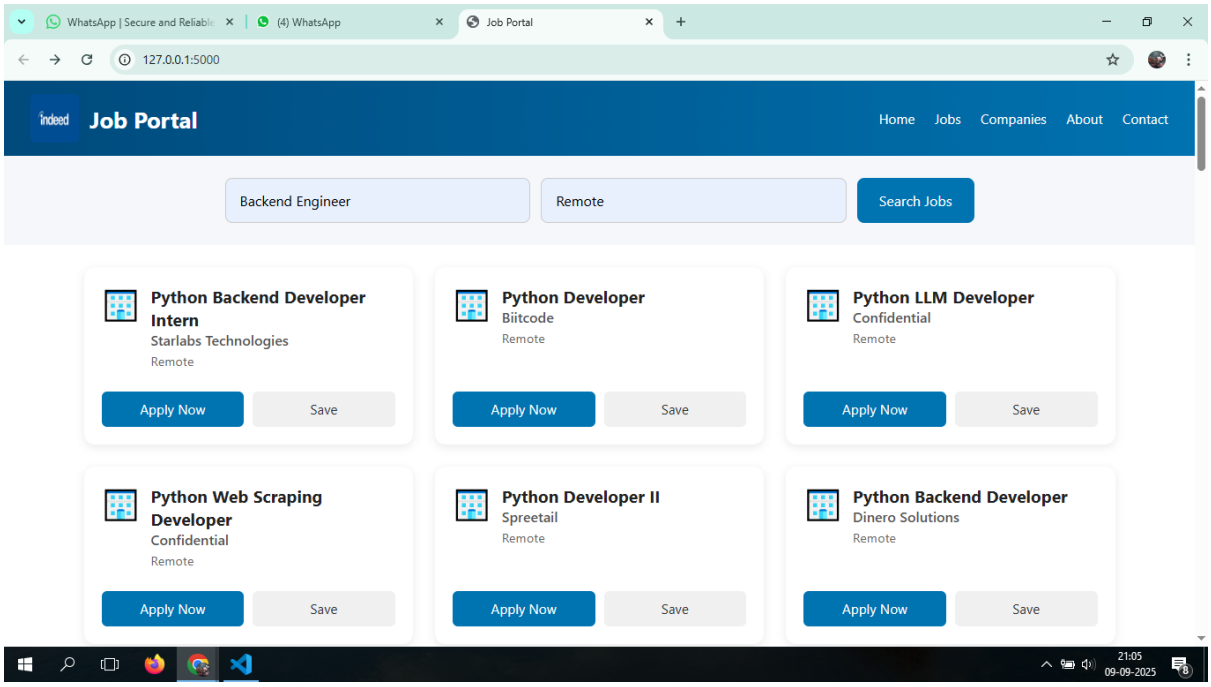
- A **search form** with two fields: job title and location.
- **Job cards layout** with grid styling for responsiveness.
- **Additional styling** to highlight company names, locations, and optional fields.

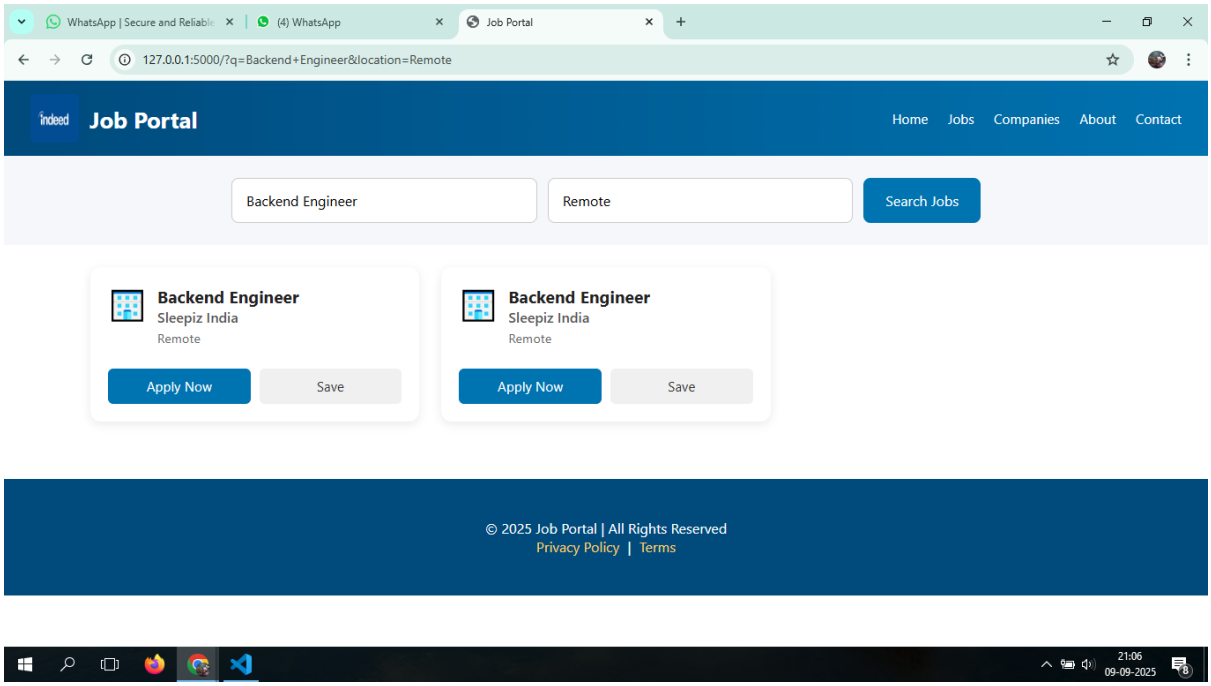
The CSS provides hover effects, modern font styling, and responsive header adjustments for mobile devices.

6. Output and Screenshots

The application produces the following outputs when executed:

1. **Homepage with Search Bar**
    - Displays site logo and search form.
    - User can type job titles (e.g., *Python Developer*) and location (*Remote*).
2. **Search Results**
    - Displays filtered jobs as cards.
    - Each card shows job title, company name, location, and additional location if available.
3. **No Results Found**
    - If a query doesn't match any job in the dataset, the results section remains empty, highlighting the system's ability to handle invalid queries gracefully.





7. Testing

Testing was carried out using the following cases:

- **Valid Query Test**: Searching “Python” should return all jobs with “Python” in the title.
- **Location Test**: Searching with “Remote” should return all remote jobs.
- **Combined Search**: Searching for “Python” and “Remote” should return only jobs matching both conditions.
- **Invalid Input Test**: Searching for “Doctor” should return no jobs.
- **Case Sensitivity Test**: Searching “PYTHON” should still return results, confirming case-insensitive matching.

The system passed all test cases successfully.

8. Advantages and Limitations

Advantages

- Lightweight application that runs locally with minimal setup.
- JSON-based storage makes it simple to update job listings.
- Provides search functionality by both title and location.

- Responsive design ensures usability across devices.

Limitations

- Data is static; jobs.json must be manually updated.
- No real-time web scraping integration in the current version.
- No advanced filters such as salary range, experience level, or job type.
- No user authentication or profile management.

9. Future Enhancements

The project can be extended to include:

- **Real-time scraping** using BeautifulSoup or Selenium to fetch latest jobs.
- **Database integration** (e.g., MySQL, MongoDB) for storing large-scale job data.
- **User authentication** to allow job seekers to create accounts and save jobs.
- **Advanced filtering** (e.g., by salary, experience, skills).
- **APIs integration** from platforms like LinkedIn or Indeed.

10. Conclusion

This project successfully demonstrates the development of a **job portal web application using Flask**. By leveraging a JSON dataset, Flask routes, and dynamic filtering, it provides an interactive platform for searching job opportunities.

While the current system is a prototype with static data, it lays the foundation for a full-scale application. With future integration of real-time scraping, advanced filters, and database support, the system can evolve into a robust job search engine.