Rohan S – RVCE24MSE003
Rohan AR – RVCE24MIT016
Vishrutha Bharadwaj-RVCE24MIT012
Akarsh R-RVCE24MIT008

Government Job and Service Portal

ObjectivesThe main goals of the Government Job and Service Portal project are:

Automated Data Collection: Automatically collect job listings and service information (e.g., welfare schemes, certificates, subsidies) from government portals.

Centralized Portal: Provide a single platform for users to access both job listings and government services from multiple sources.

Real-Time Updates: Schedule scraping to ensure both job postings and service details are up to date.

Search & Filtering Functionality: Enable users to search by keyword, department, location, eligibility, job type, and service category.

JavaScript-Rendered Scraping: Use Selenium to extract dynamically loaded content for jobs and services from government websites.

Data Storage & Caching: Use Redis or Memcached to store frequently accessed job and service data, reducing database queries and improving performance.

Duplicate Data Prevention: Prevent redundancy by checking for existing job and service entries before storing new ones.

User-Friendly UI/UX: Design a responsive and intuitive web interface using Bootstrap for seamless navigation of jobs and services.

Technical Details

Technologies Used

- **Frontend:** HTML, CSS, Bootstrap
- **Backend:** Django (Python)
- **Data Storage: Caching Mechanism** (e.g., Redis, Memcached) for faster job retrieval
- **Web Scraping:** BeautifulSoup, Requests, **Selenium** (for handling JavaScript-rendered content)
- **Version Control:** Git/GitHub
- **Hosting (Op:** AWS)

Workflow

Phase 1: Django Project Setup

- **Create a Django project and apps for job listings and services.**
- **Configure the caching mechanism.**
- **Set up a basic HTML frontend for jobs and services.**

Phase 2: Scraping Data

- **Jobs:**
  - **Use BeautifulSoup and Requests to scrape job postings from government websites.**
  - **Extract relevant details: Title, Department, Location, Last Date, and Apply Link.**
  - **Store scraped job data in caching.**

- **Services:**
  - **Use BeautifulSoup and Requests to scrape service details from government portals.**
  - **Extract relevant details: Service Name, Eligibility, Category, Department, and Application Link.**
  - **Store scraped service data in caching.**

Phase 3: Displaying and Searching

- **Fetch jobs and services from caching and display them on the website.**
- **Jobs:**
    - **Implement search and filtering based on Title, Location, and Job Type.**

- **Services:**
    - **Implement search and filtering based on Service Name, Eligibility, and Category.**

Phase 4: Automation and Optimization

- **Implement periodic scraping for jobs and services using Django management commands or Celery.**
- **Avoid duplication of job and service data in the database.**
- **Optimize search and filtering functionality for better performance.**

Methodology

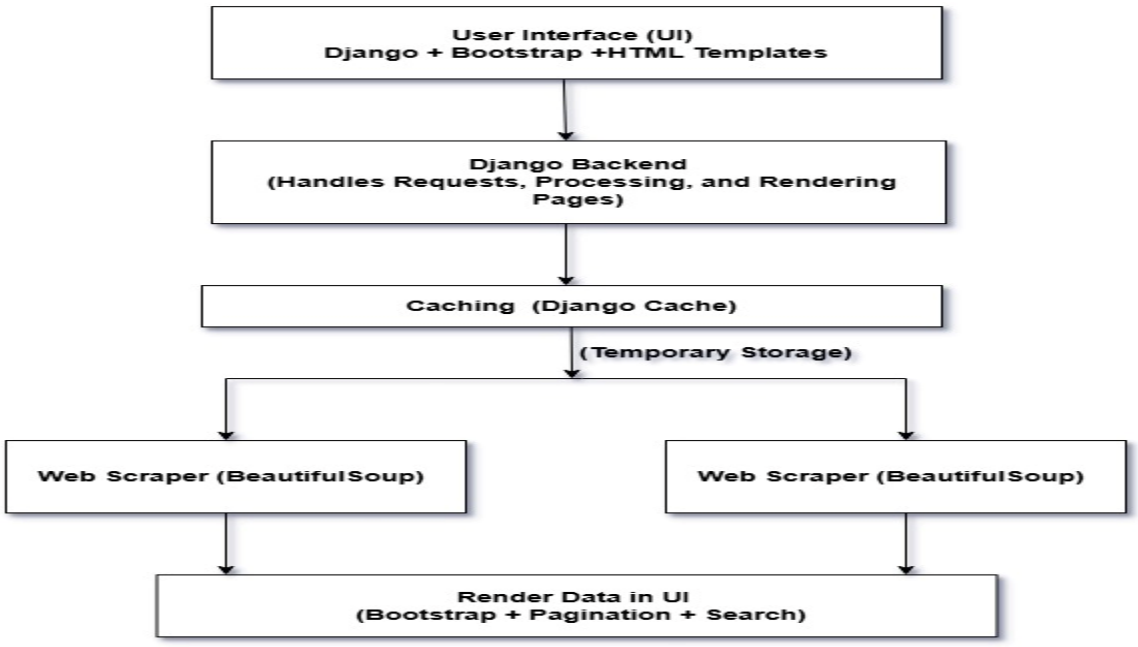

Fig. 1. Architecture Diagram

Step 1: Data Collection (Scraping Government Job Websites)

- Use Python's **Requests** and **BeautifulSoup** to fetch job listings from government job portals.
- Extract **Job Title, Department, Location, Last Date, and Apply Link** from the website.
- Store the extracted data temporarily for processing.

Step 2: Data Processing (Cleaning and Structuring Data)

- Filter out **duplicate job postings**.
- Ensure **uniformity** in job titles and locations.
- Validate the **apply links** to make sure they are correct.

Step 3: Data Storage (Database Integration - Caching)

- Store the cleaned data into a **Caching Mechanism.**
- Use Django **models** to define database structure.

Step 4: Data Retrieval (Fetching Jobs for Website Display)

- Fetch stored jobs from the **Caching**
- Display them on the **Django-based website**.

Step 5: User Interaction (Search & Filters Implementation)

- Users enter keywords or filter jobs by **category & location**.
- System retrieves relevant jobs **in real-time**.

Step 6: Automation (Periodic Scraping & Updates)

- Implement a **cron job or Django management command** for automatic data updates.
- Ensure old job postings get **removed automatically**.

Prototype Analysis

Initial Prototype:

Fig. 2. Initial Prototype

The initial prototype focused on setting up the **Django framework** and implementing basic **web scraping** to extract job listings from government websites. The scraped data was displayed in a simple **HTML page** without advanced filtering or search functionality. The key features in this phase included:

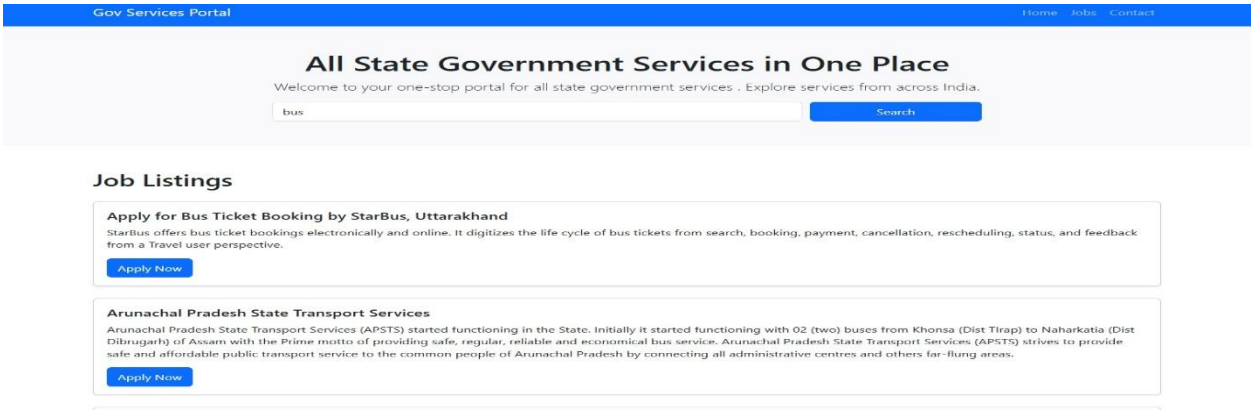

Fig. 3. Initial Prototype after Scraping

- Basic **job data extraction** using **BeautifulSoup** and Requests.
- **Static job display** with no search or filtering options.
- **Manual scraping execution**, requiring a user to trigger updates.

- Basic **Caching Mechanism** for storing job listings.

Final Prototype:

The final prototype introduced **significant enhancements** to improve automation, usability, and search efficiency. Key improvements included:

- **Selenium Integration:** Enabled scraping of **JavaScript-rendered job listings**.
- **Search and Filtering Functionality:** Users could now **search jobs by keyword** and filter based on criteria like **location and job type**.
- **Dynamic Job Display:** The job listings were retrieved **from the database dynamically**, ensuring real-time updates.
- **Automated Scraping:** A **periodic scraper** was implemented using Django management commands, eliminating the need for manual scraping.
- **Improved UI/UX:** A **Bootstrap-based responsive interface** was developed for better accessibility and navigation.
- **Error Handling & Optimization:** Implemented **duplicate job prevention, timeout handling, and structured data storage** for better performance.

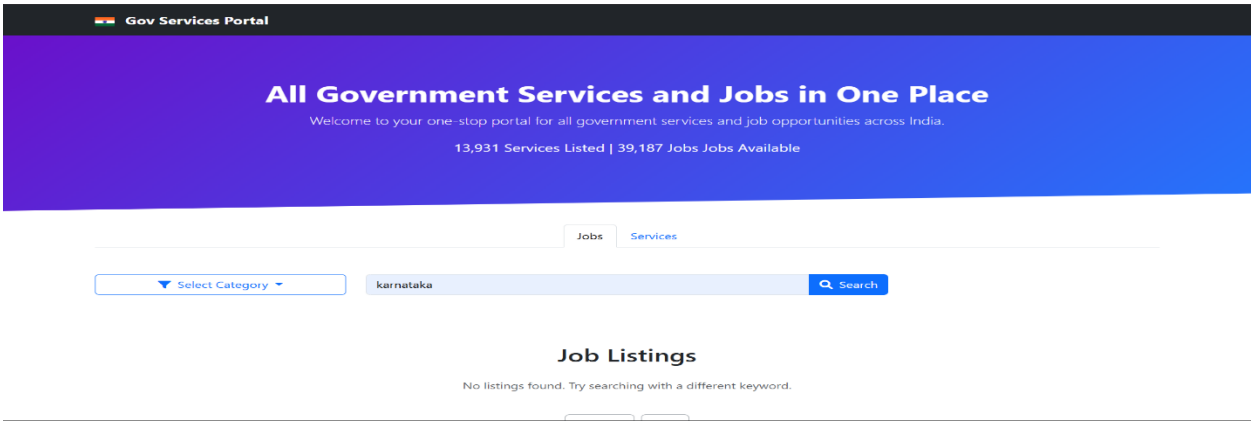

Fig. 4. Final Prototype

Fig. 5. Final Prototype after Scraping

Results and Observation

- **Successful Job Data Extraction:**
    - The system successfully scrapes job listings from government websites.
    - Extracted details include **Job Title, Department, Location, Last Date, and Apply Link**.
    - Using **Selenium**, the scraper handles JavaScript-based job listings effectively.
- **Efficient Data Processing and Storage:**
    - The cleaned and structured job data is stored in an **SQLite database**.
    - Duplicate job postings are removed before insertion.
- **Dynamic Job Display on Website:**
    - The job portal dynamically fetches job data and displays it to users.
    - Users can search and filter jobs based on **keywords, job type, and location**.
- **Challenges Encountered:**
    - **Anti-scraping measures:** Some websites blocked direct requests, requiring **headers and Selenium** to bypass restrictions.

- **Frequent website structure changes:** Scraping logic needed periodic updates.
- **Performance issues:** Direct real-time scraping caused delays, solved by implementing periodic background scraping.

Fig. 6. Jobs Listed with Associated State Names

Fig. 7. Job Listings Filtered by Category

Fig. 8. Service Results Displayed in a Different Tab

Inference

- The project successfully demonstrates how web scraping can be used to aggregate government job postings.
- The implementation of Django allows for efficient data management and display.
- Periodic updates ensure job listings remain current and relevant.
- The search and filtering features improve user experience by making job searches more efficient.

Future Enhancements

- **Database Optimization:** Move from SQLite to PostgreSQL for scalability.
- **Advanced Scraping Techniques:** Implement Selenium for scraping websites with JavaScript-based content.
- **User Authentication:** Allow users to sign up, save jobs, and receive job alerts.
- **Job Alerts System:** Send email notifications to users based on their saved job preferences.
- **Mobile Responsiveness:** Improve UI/UX for mobile users.
- **AI-Based Job Matching:** Recommend jobs based on user profile and past searches.

Conclusion

The development of this **Automated Government Job Portal** successfully addresses the challenge of manually searching for government job postings across multiple websites. By leveraging **Django** for backend development, **Selenium and BeautifulSoup** for web scraping, and **SQLite** for data storage, the system efficiently collects, processes, and displays job listings in a user-friendly manner. The integration of **search and filtering functionalities** enhances accessibility, allowing users to quickly

find relevant job opportunities. One of the key achievements of this project is its ability to handle **JavaScript-rendered job listings** using Selenium, ensuring that even dynamic content can be extracted and displayed in real-time. Additionally, the implementation of **periodic scraping** ensures that job postings remain up to date while preventing redundant entries.

Despite facing challenges such as **anti-scraping measures and frequent website structure changes**, the project successfully demonstrates the power of **automation and data aggregation** in streamlining the job search process. Future improvements could include migrating to a more **scalable database like PostgreSQL**, implementing **user authentication for personalized job alerts**, and integrating **AI-powered job recommendations** to enhance the user experience. By automating job data extraction and improving search efficiency, this project serves as a robust foundation for a **smart, centralized government job search platform**, ultimately saving users time and effort in finding suitable employment opportunities.

Significance of the Work

The Government Job and Service Portal platform is significant for several reasons:

- Provides a centralized platform for government job seekers.
- Automates the tedious process of searching for jobs across multiple websites.
- Helps users find relevant job opportunities quickly and efficiently.
- Demonstrates the practical use of web scraping and database management.
- Serves as a foundation for future improvements in job aggregation and recommendation systems.